Technische Universiteit
Eindhoven
University of Technology

Department of Mechanical Engineering
Control Systems Technology

Systems engineering languages for modeling and analyzing supervisory control structures in cyber-physical systems

Master's Thesis

Antoni Planells Valencia

Supervisors:
Michel Reniers
Bram van der Sanden

Eindhoven, July 2016

Abstract

In today's world, a new generation of high-tech cyber-physical systems are becoming an integral part of our societies and their impact is only going to increase within the next years. Because of their importance, the companies that develop these systems use proper systems engineering modeling tools to help with the design and development of these types of systems and to accelerate the whole development process.

In this thesis, 4 very popular modeling tools/languages are being tested and evaluated in terms of their capabilities for model-based systems engineering. These tools are Simulink&Stateflow from MATLAB, Modelica, MechatronicUML and SysML. In order to do that, a proper introduction of the systems engineering process is presented to set the criteria in which the different tools/languages will be evaluated. To support the evaluation process, a case study is presented with the CIF3 language that will be attempted with all the other languages/tools. Each modeling language/tool has been evaluated individually at first and then together with the others in the end. In addition to the first evaluation, a proper basic introduction of all the modeling concepts that each tool uses for modeling cyber-physical systems is provided and the building of the case study as well. After that, in the second evaluation, the languages are extensively compared against each other in terms of all the criteria set previously to see exactly the scope of capabilities that each tools has. As a result from the two evaluations, a definitive review for each language/tool is presented addressing their overall scope of capabilities, main strong features, main uses, possible ways of improving and future development.

Preface

This thesis is the culmination of a journey that I started six years ago and that has been one life-changing experience. I started my bachelor degree in Industrial Engineering at the School of Industrial Engineering of Barcelona (ETSEIB) in 2010. I have always had curiosity towards mathematics and all its possible applications in the industry and that is why I thought my best way to pursue that would be to do that bachelor program. I will never regret making this decision. During that time I devoted quite an interest in engineering fields like robotics and automatic control. After the bachelor, in 2014, I continued my studies with a Master in Industrial Engineering at the same university an I specialized in Robotics and Automatic Control. During this period of time, I had the oportunity to end my Master's Degree at the University of Technology of Eindhoven (TUe), at the Mechanical Engineering Department, and to do my thesis there. This project was done from the 1st of February till now in the Control Systems Technology research group and has opened my eyes to some engineering fields that I was not aware of and that have caused a lot of interest in me.

This thesis would not have been possible without the support, help and advice of many people. First of all, I would like to thank my supervisors Michel Reniers and Bram van der Sanden for giving me the opportunity to work on such an interesting project with them. Their strict guidance combined with their spontaneous enthusiasm has helped a lot during this process and has kept me always on the right track. This experience has been of great value for me, as an engineer and as a person, and I will always be grateful for that. I would also like to thank all the people that I have met during my stay here Eindhoven and that have treated me well making me feel welcome like at home. This includes the research students from my lab and all the Erasmus students that have had experiences with me during this months. I would also like to thank my friends from back home that have kept in touch with me despite the distance and that have supported me during this whole process. Finally, I would like to express my gratitude towards my parents, sister and family for their unconditional support during this experience and my whole life. It means the world to me.

After this thesis, when I am done with the Master, I will start a new chapter in my life and I am really looking forward to it. I feel ready to face all the challenges that life will put in my way and I am mentally ready to overcome them and never give up on my dreams. This years have made me grow from the little boy I was when I started the university to the grown man that I am now. I would not be here where I am without all the people that I have encountered so I say once again: Thank you.

Antoni Planells Valencia

July 20, 2016

Contents

Chapter 1

Introduction

1.1 Cyber-physical systems

In today's world, computing and communication devices are becoming smaller and cheaper resulting in embedded objects and structures that interact directly with the physical environment and extend the humans capabilities. This phenomenon has resulted in a new generation of systems with multiple sensing and actuation units that allows them to gather, process, exchange and use information in order to bridge the cyberworld of computation and communication with the physical and biological world. These systems are known as Cyber-Physical Systems (CPS), see [43].

The cyber-physical systems are systems that link the cyberspace with the physical world through a network of interrelated elements such as sensors and actuators, robots and manipulators, and computational engines. Embedded computers and networks are used to monitor and control the physical processes, normally by using feedback loops where the physical processes affect the computations and the other way around. In order to improve the scope of their capabilities, the future technology developments in the area of CPS are mostly focused on the ability that we can have to interact with the physical world through computation, communication and control trying to expand their capabilities to the optimal limits. This generates new research challenges and opportunities which include the design and development of next-generation airplanes and space vehicles, hybrid gas-electric vehicles, fully autonomous urban driving and prostheses and other medical devices that allow brain signals to control physical objects. Apart from these examples of CPS you could also consider robotic systems, manufacturing systems and smart spaces like common CPS that you find in today's world.

Regarding CPS impact nowadays, as time goes by, cyber-physical systems are becoming an integral part of modern societies even though the first ones appeared some decades ago. In the early 1970s, for example, the automotive embedded systems did already combine a closed feedback-loop control of the brake and the engine subsystems (physical parts of the CPS) with an embedded computer system (cyber parts of the CPS). Since then, new functionalities, requirements and networking have dramatically increased the scope, capabilities and complexities of any CPS. This has created some urgent needs to bridge the gaps between separate CPS sub-disciplines such as computer science, automatic control or mechanical engineering in order to establish CPS as an intellectual discipline in its own right.

In terms of the design, cyber-physical systems are challenging [18] because:

- The vast network and information technology environment connected with physical elements involves multiple domains such as controls, communication and logic.
- The interaction with the physical world varies widely based on time and situation.

Because of this, the usage of multi-domain models that capture such variability is critical to successful CPS development. Figure 1.1 shows some of the different domains that any CPS has and that have to be taken into account.

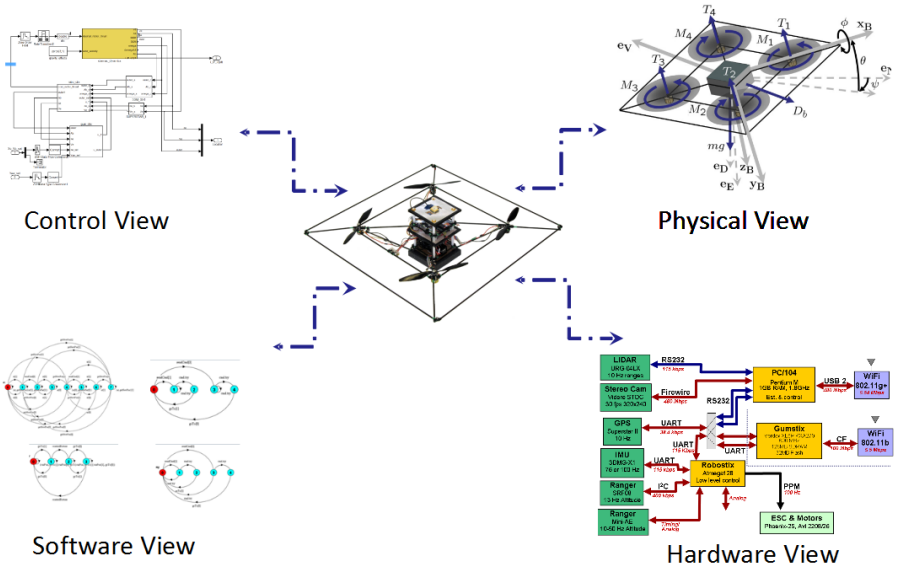

Figure 1.1: Multi-domain models of a CPS [7].

In the development of a CPS many entities (we can call stakeholders or contributors) are involved and each one them has different interests in different parts of the system. This results in different persons (stakeholders) assuming different roles regarding certain problems or questions that involve multiple engineering disciplines. In the resolution of all the problems and questions that have to be addressed and solved, modeling plays a key role in the development of the whole system.

During the design process many modeling languages and tools are used to reflect the heterogeneity of the CPS. The use of models during the design process of a CPS is called model-based system engineering [22]. The use of this practice leads to multiple models (of different aspects of the system) captured in different modeling tools which might have unclear relations among them. For instance, during the control design stage an engineer may use continuous-time models to model the plant and the controllers just to later convert those models to discrete-time models so that they can be used for code generation. On the other hand, software engineers might probably use various Unified Modeling Language (UML) [46] diagrams to design different aspects of the software part of the system which will be integrated later with the code generated controllers. Therefore, a proper selection of modeling tools is key during the design stage and can bring a lot of benefits if they can model several aspects of the systems and their respective communications and relations.

1.2 Systems engineering

Systems Engineering is an approach that combines many different engineering disciplines and that has as main objective the development and realization of successful cyber-physical systems. This approach integrates all the disciplines and specialty groups into a combined effort creating a structured development process from concept to production to operation. It considers both the technical and the business needs of all the customers/stakeholders with the goal of creating a product that satisfies the user's needs.

This engineering process, as it can be seen in [35], can be divided in the following seven steps that can be coupled in the acronym SIMILAR. This steps are not exclusive and can be performed in a parallel manner:

- **State the problem**. The process starts with a description of the top-level function that the system has to perform in order to be accepted. This can be expressed in the form of a mission statement or a description of the deficiencies in the system that must be improved. This description should express the stakeholder's requirements in functional or behavioural terms (either as a text or as a model). Common stakeholders are the end users, maintainers, suppliers, operators, owners manufacturers and other customers.

- **Investigate Alternatives**. During the design phase, alternative designs are created and evaluated based on their performance, schedule, cost and risk figures of merit. Since almost no design is likely to be the best in all the indicators, to choose the most convenient alternatives multi-criteria decision-aiding techniques should be used in order to help with the process. This analysis of alternatives is a continuous process that is redone as soon as more data is available to study. Simultaneously, models are constructed and strictly evaluated so that later on the prototypes are built. After that, several tests have to be run on the real system to ensure the desired functionality.

- **Model the system**. For each of the alternative designs presented, models will be developed and once the preferred alternative is chosen this will lead to an expanded model to help manage the system through the entire life cycle. The types of models to be used include block diagrams, object-oriented models, state machines, data-flow models, analytic equations and mental models. Models should not only be constructed for the final product but also for the process to produce it since systems engineering is responsible for creating a product and its production process.

- **Integrate**. To integrate means to bring some entities together so that they work together as a whole and that is what systems, businesses and people do in order to interact with one another during the development process. Subsystems must be defined along natural boundaries and to minimize the amount of information to be exchanged between them. For these subsystems, some interfaces must be designed between them so that a system is built and operated through efficient processes.

- **Launch the system**. In this phase, the preferred alternative previously chosen is designed with a high degree of detail. This includes the parts that are built by the company or bought and the integration and test of the different parts which leads to a certified final product. This process of designing and producing the system is iterative and can suffer modifications along the way since new knowledge is developed that may cause a re-consideration of the steps in the development process.

- **Assess performance**. To assess the performance of the system in certain situations indicators like figures of merit, technical performance measures and metrics can be used. The process of measurement is key because if you cannot measure something you cannot control it and if you cannot control it you cannot improve it. Figures of merit are usually used to specify the requirements in the trade-off studies, the technical performance measures are employed to reduce risk during the development process and metrics are used to manage the company's processes.

- **Re-evaluate**. This is probably the most important task among all of them since is one of the most fundamental systems engineering tools. This should be a continuous process with many parallel loops that consist in observing outputs and using this new information to modify the system in a matter of its inputs, the product or the process.

The above description of the Systems Engineering process is just a proposed model which may not coincide with all the cases. However, in a lot of cases the systems engineering process will share

a lot of similarities with this one. A summary of the systems engineering process, also referred to as the V-model [38], can be seen in Figure 1.2.

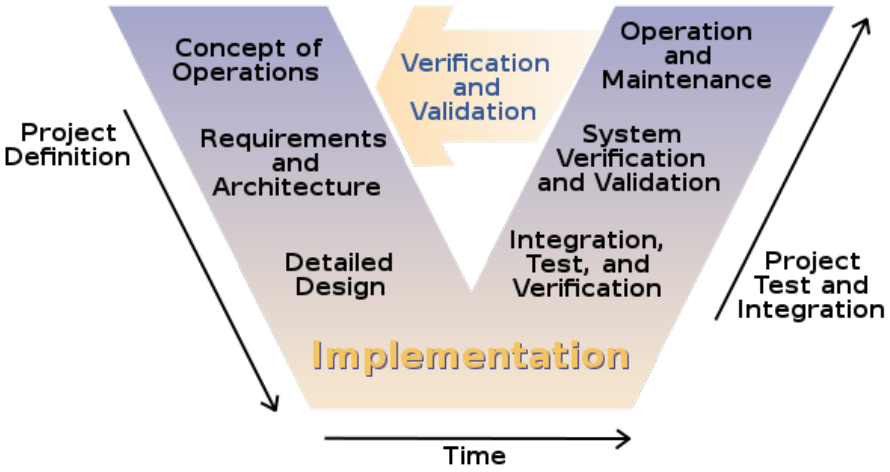

Figure 1.2: V-model [38].

1.3 Model-based engineering

Model-based engineering (MBE), also known as model-driven engineering (MDE) (since the terms model-based and model-driven are commonly merged), is a software and systems development methodology which focuses on creating and exploiting conceptual models of different disciplines which are related to a specific problem.

This approach has as main purpose to increase the productivity by maximizing compatibility between systems through the reuse of standardized models, simplifying the design process and promoting the communication between teams working on the system to give better feedback and evolve faster.

This MBE methodology is considered effective if any user of the application domain understands properly the generated models and if these models can be used as the base for the later implementation of the system. These models are developed through extensive communication among product managers, designers, developers and users of the application domain. As these models approach completion, they enable the development of software and systems.

Some of the best known MBE initiatives are:

- the Object Management Group (OMG) initiative of the model-driven architecture (MDA), which is a registered trademark of OMG [36].
- the Eclipse ecosystem of programming and modelling tools (Eclipse Modeling Framework) [49].

Model-based engineering is an “umbrella” term that subsumes several sub-disciplines like model-driven development (MDD), which focuses on software-intensive applications, and model-based systems engineering (MBSE), which focuses on systems engineering applications among many others.

1.3.1 Model-based systems engineering

Model-based systems engineering (MBSE) [22], also known as model-based systems development (MBSD), is a methodology which focuses on the application of precise modeling principles and practices to the activities in the systems engineering process throughout all the system development life cycle (SDLC) [42]. These systems engineering activities include, but are not limited to, requirements analysis and verification, functional analysis and allocations, performance analysis, trade off studies, and system architecture specification [57].

MBSE enhances the ability to capture, analyze, share, and manage the information associated with the complete specification of a process, resulting in the following benefits:

- Improved communications among all the development stakeholders (e.g. the customer, program management, systems engineers, hardware and software developers, testers, and specialty engineering disciplines).
- The possibility to model a system from several different perspectives (and investigate/analyze alternatives on them) allows to deal/manage the complexity that some high-tech CPS might have.
- By providing evaluation possibilities for correctness, completeness and consistency (among others), the developed systems have improved quality and less risk of failure.
- Enhanced knowledge capture and reuse of the information by capturing information in more standardized ways and leveraging built in abstraction mechanisms inherent in model driven approaches [44]. This in turn can result in reduced cycle time and lower maintenance costs to modify the design.

1.4 Cyber-physical systems development process

The development of cyber-physical systems always requires a strong interaction between different engineering disciplines like mechanics, electronics and control during the whole development process. This is quite a complex process and since the functionality and performance of these systems always depends on the interaction between the control software and the hardware resources this is something that has to be taken a lot into account.

The development process, including the control system, starts with an extensive analysis of the requirements from the different users/stakeholders which describe the desired functionality and performance for the real system. From these requirements, conceptual specifications regarding the hardware and control components are derived during the concept development phase. Using these requirements as a start, the machine's dynamics and the control strategy to control it are defined with the help of models which are properly developed, simulated and analyzed. It is very important to emphasize that in this stage the primary objective is that the stakeholders requirements are fulfilled. These new specifications are used as guidelines for the next stages of the development process and this implies that it is very important that the specifications are correct. By correct, we mean that the hardware and the control system work according to their corresponding specifications so that the system fulfills the mandatory requirements.

During the concept-development phase, a very powerful tool like simulation can be used to determine, define and validate all these specifications for the system. Since this tool can be used from the early stages in the development process this leads to a reduction of concept-design errors thanks to simulation-based validation. The physical-design phase is used to define with a high degree of precision how the control system interacts with all the actuators and sensors from the hardware system. This leads to the creation of several models with extensive information about the sensors

and actuators of the system, and the control system is adapted to guarantee the desired overall performance of the system.

In the last implementation and testing phase, the control system is implemented in the real-time platform. For desired functionality in real-time, the calculations are better performed within well-defined time periods. Therefore, it is important to evaluate the impact of the calculations durations on the communication behaviour. In case of any differences with respect to the behaviour seen in simulation time, an extensive correctness analysis has to be done and this can lead to changes in the previous design. In the end, the control system operates the real machine through an I/O interface.

The first two phases in the development process give virtual system models which are reliable representations of the systems components and the control system that is associated with them. Ideally, in the implementation and testing phase, these virtual system models are replaced by the real system and the control system is applied in a real-time environment without suffering more changes in its design.

1.5 MBSE and supervisory control

The model-based systems engineering (MBSE) methodologies provide various advantages for supervisory controller development. Concretely, formal and executable models are built and employed in the design phase for design-space exploration of the supervisors that control the system. These models are a crucial part of the design process, since all subsequent steps such as the synthesis, validation, verification, testing, and code generation all depend on them. Therefore, it is very important that the models are easy to create, understand, and adapt.

Supervisory Control Theory (SCT) [8] typically looks at systems modeled as a discrete-event system (DES) [12]. DESs are discrete-state, event-driven systems, whose state evolution depends entirely on the occurrence of asynchronous discrete events. When designing the supervisory control functions for DESs, model-based approaches can be used to understand the systems behavior. The most commonly used approach in order to model the behaviour of the system is automata, also known as Finite-State Machines (FSMs).

This approach is also used in the synthesis-based MBSE framework [3], where FSMs are extended with variables resulting in extended finite automata (EFAs) [32]. In this framework, the first step consists of the creation of discrete-event models of the uncontrolled system that is often referred to as plant. Then, the requirements regarding the control of the system are modeled. These requirement models might refer to variables and states (also of other automata) to enable concise modeling. After modeling, Supervisory Control Theory can be applied to automatically synthesize a control function, referred to as supervisor. The supervisor is used to restrict the behaviour of the plant such that the system never violates any of the given requirements and constraints. The main feature of the synthesized supervisor is not to restrict the behaviour of the plant more than strictly necessary while guaranteeing that the control requirements are achieved. This is referred to as minimally restrictive supervisor.

1.6 xCPS platform

The cyber-physical system that has been chosen to be studied (or at least part of it) in this thesis is the xCPS platform [1]. The xCPS platform (Figure 1.3) is a small scale machine that mimics a production line capable of assembling and disassembling objects. The cylindrical assembly pieces of the xCPS (which can be seen in Figure 1.4) come in two complementary shapes (top work piece and bottom work piece), and three colours (silver, black or red).

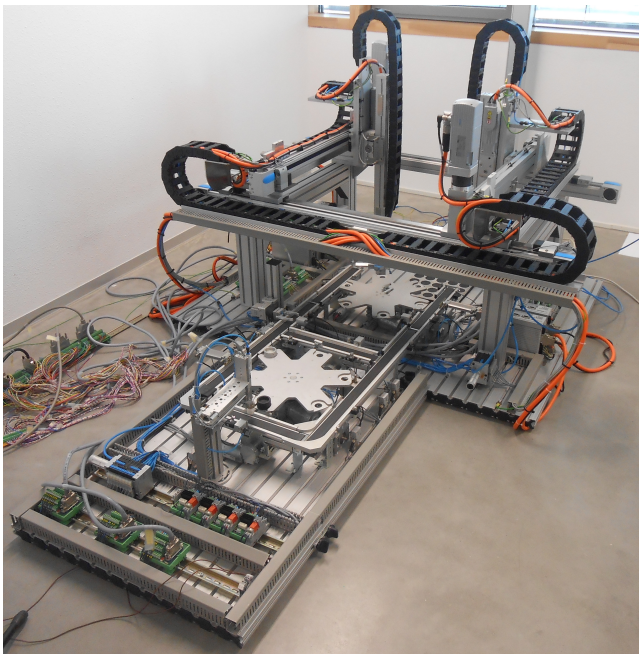

Figure 1.3: xCPS Platform [23].

High-tech Cyber-Physical Systems such as xCPS provide bachelor/master students and PhD researchers with a realistic platform to perform measurements and analysis on, hence enabling development of novel techniques that are closer and more applicable to real practice.

Figure 1.4: Assembly pieces created in the xCPS platform [1].

The system layout (Figure 1.5) illustrates the main components of the platform. The platform consists of a storage area, six conveyor belts, two indexing tables, two gantry arms, and several actuators and sensors. The storage area is a grid where up to 25 components can be stored.

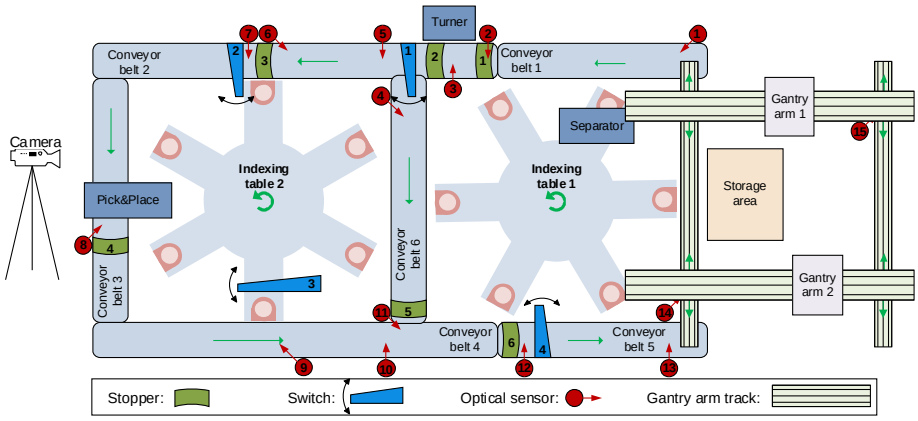

Figure 1.5: xCPS system layout [1].

In this platform there are six actuators called "stoppers" in strategic positions along the assembly process that can obstruct the movement of pieces, creating buffers accumulating pieces on the respective conveyor belts as a result. Other actuators, called "switches", allow to change the route of individual objects and there is also a turner that can flip pieces if they are not well oriented. The platform also contains two actuators for assembling and disassembling of pieces. Then there is a pick&place actuator whose function is to clamp a part and combine it with a complementary part (a top work piece with a bottom work piece). A separator can disassemble two combined pieces. The xCPS platform is equipped with 15 sensors that can detect the presence of an object in the surrounding area. These sensors cannot distinguish the type of an object or its colour and in order to detect the type, colour, and location of an object, a camera is added to the set-up.

Since this is a very large system (in terms of sensors, actuators and components) to try to model in order to test the different tools capabilities, first a reduction had to be made resulting in a smaller system that still had all the features of a high-tech CPS. That narrowed our focus to the assembling area (Figure 1.6) of the platform and specifically to the Pick&Place station.

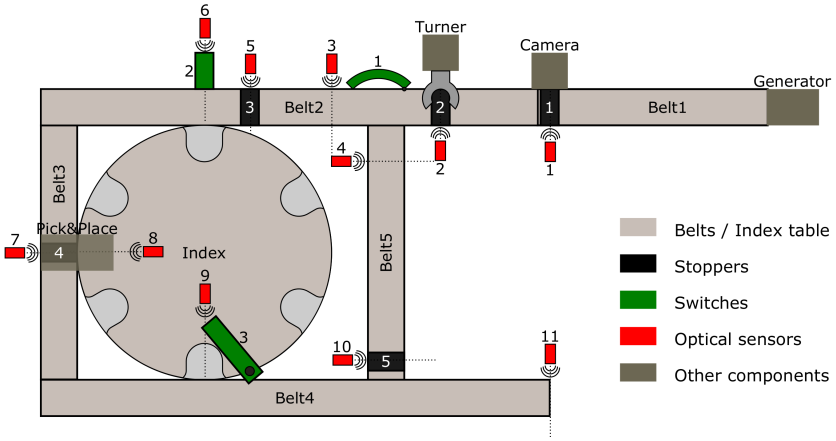

Figure 1.6: A top view of the assembling and transporting part of the flow shop workstation. [23]

The Pick&Place unit is composed by three actuators and five sensors and its main function is to pick top work pieces and drop them on the bottom work pieces that arrive from the index table. Some models of this station were created in [23] with the Compositional Interchange Format 3 (CIF3) tool set [50] and will be used later in this thesis.

1.7 Objectives and outline

The main objective for this thesis is to analyze and compare the capabilities and possibilities of certain languages and tools for modeling and analyzing the supervisory control structures in the systems engineering processes to develop cyber-physical systems. The tools that are going to be compared will be the following:

- Simulink&Stateflow [52, 53]
- Modelica [25]
- SysML [24]
- MechatronicUML [4]

The idea is to select a set of concepts that are the most relevant in the different phases during the development process of systems with supervisory control synthesis and to see if the different modeling languages can implement them. These concepts will be based on the CIF3 toolset since it is a very powerful tool in terms of modeling MBSE processes with supervisory control synthesis. This will lead to a comparison where we will see what each tool is capable of modeling/analysing and the process that they have to follow to do it. This way we will be able to determine which tools are more powerful when trying to develop cyber-physical systems with supervisory control structures. A more extensive explanation on the methodology that will be used in this thesis to compare the different languages will be presented in the next section.

To support this comparison, a case study will be presented with the modeling process of the Pick&Place station that has been previously introduced in Section 1.6 followed by some simulation results. This modeling process will be tried with each tool with the objective of creating a model with the same functionality (probably with different structure or semantics) as the one modeled with CIF3. The simulations will also tried to be run and a comparison of the results that are obtained will be done.

Therefore, this thesis will have the following structure:

- In Chapter 2, model-based systems engineering and discrete-event system will be introduced. This will include the fundamentals of model-based systems engineering with supervisory controller development, behavioural models, control strategy and the models for supervisory control.
- In Chapter 3, the case study that has been chosen for this thesis will be explained. To do that, first of all, we will introduce the modeling tool CIF3 that has been used to build the case in the first place. The modeling process will be explained in detail from modeling the plants and requirements till the simulation results with the supervisory controller. Eventually, an evaluation of CIF3 capabilities in terms of model-based systems engineering with supervisory control synthesis will be done just like it will be done with all the other tools.
- In Chapters 4, 5, 6 and 7, the modeling languages/tools Simulink&Stateflow, MechatronicUML, Modelica and SysML will be introduced and evaluated in terms of their capabilities in model-based design. All these chapters will have the same structure. First of all, the modeling language/tool will be introduced. Next, an introduction of the important syntactic components that the language uses for modeling will be provided. After that, the evaluation of capabilities will be done with criteria that will be defined for all the tools and that is based on a set of fundamental concepts. Finally, each chapter will end with the modeling process of the case study that has been chosen for this thesis along with simulation results and its analysis.

- After having evaluated the modeling tools separately, in Chapter 8 a comparison between all the languages will be done to assess the capabilities of the modeling tools w.r.t. each other. The comparison will be done with a table with all the concepts that are considered important for model-based systems engineering with supervisory control development and with all the languages to see which language provides which concepts. Qualitative explanations regarding the concepts will also be added in order to be capable to compare the methods that are used in each language and their real capabilities and limitations. After the comparison, a review will be performed to sum up the capabilities of each tool and to highlight which are the features that outperform the other languages/tools.
- In Chapter 9, the conclusions of the work done in this thesis will be presented. This chapter will also include some recommendations and hopefully some guidelines for future work that could be done in relation to this thesis.

1.8 Methodology for comparing and evaluating modeling languages

As mentioned in the previous section, the main goal of this thesis is to do an extensive comparison between different modeling languages to see their capabilities in development of cyber-physical systems. This comparison will be performed based on a set of features/concepts that are the most relevant in model-based systems engineering with supervisory control synthesis. This MBSE process can be divided in 4 main phases and the concepts that will be tested in each phase are the following:

- Specification phase
  - Plant modeling
    - Time-driven
      - Continuous-time
      - Discrete-time
    - Event-driven
      - Automata
      - Mode switching
      - Events
      - Final states
      - Updates of variables during time
      - Discrete variables
      - Synchronization on time
      - Synchronization of events
      - Shared variables
      - Initialization
  - Requirement modeling
- Supervisory development phase
  - Manual development
  - Synthesis algorithm
- Validation phase
  - Simulation-based visualization
  - Formal verification

- Implementation phase
  - Code generation

Chapter 2

Introduction to Model-Based Systems Engineering and Discrete-Event Systems

This chapter is partly based on [3] and [35].

In today's world, developing high-tech cyber-physical systems is very difficult because of certain factors like the system's complexity or resource limitations. To overcome these difficulties, the use of models is increasing during the whole development process. In the design phase formal and executable models are built and employed to assess functional correctness and performance of each one of the components and also for the overall system. When a high degree of confidence in the functional correctness of the design is required formal verification is usually used, in particular model checking. In terms of assessing design performance, people usually use simulation directly on the model.

Figure 2.1 shows the typical control architecture of a high-tech system. This architecture can be divided in two subsystems, the hardware and the control system. The first one is located at the lowest level and consists of the uncontrolled physical components. The control system is subdivided into low-level and high-level control. The low-level control usually operates in the continuous, time-driven domain and in this layer the controlled system is observed and actuated through sensors and actuators, respectively. The control signal for the actuators is derived from high-level commands and low-level control loops. On the other hand, the high-level control normally operates in discrete, event-driven domain and in this layer the monitored sensor signals are processed and combining them with the user inputs, the low-level components are coordinated by the high level-controller which is referred to as supervisor or supervisory controller (S).

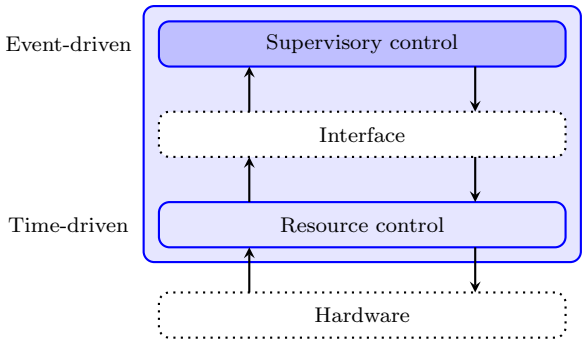

Figure 2.1: Control architecture of a high-tech system [35].

To include supervisor control design in model-based engineering, new modeling and analysis techniques are requested in order to handle the complexity that is usually faced in high-tech systems and to support the adaptive evolution of the development process. On top of this, high-tech companies are usually requested to increase the quality and functionality of their product while reducing cost in matters of time-to-market and production. As a result, there is a need for new systems engineering processes.

This section is structured as follows. Section 2.1 introduces the fundamental theory of the model-based systems engineering approach with supervisory control synthesis. In Section 2.2 we briefly explain the different type of models that can be used in systems engineering processes in order to improve and speed up the development process. After introducing all the different models, in Section 2.3 a few general control strategies which relate to continuous/discrete-time and discrete-event systems will be introduced and briefly defined. Eventually, Section 2.4 gives an overview of the models that are essential for supervisory control development along with the relevant process steps and relationships between models.

2.1 Fundamentals of MBSE with supervisory control synthesis

As previously stated, the industry nowadays faces an increase in complexity and is constantly challenged in reducing cost and development time [18]. To confront these challenges, a next generation of MBSE approaches was introduced in order to reduce human mistakes and compresses the development cycle. The MBSE method proposed in [3] enables the use of several analysis techniques and tools that are based on formal models to support system development. The reduction in cost and cycle time can be gained with the incorporation of methods and tools for supervisor synthesis.

Synthesis [40] is a technique where a plant model and a requirement model are combined into a supervisor which, when composed with the uncontrolled plant, has the properties expressed by the defined requirements. These requirements are divided in terms of supervisory control (high-level control) and regulatory control (low-level control). This new procedure brings an automatic derivation of the control logic, replacing the manual design in the process, from the components (plants) and requirements specifications. The resulting synthesized control logic is non-blocking and correct with respect to safety properties. The position of these supervisor synthesis techniques in the engineering development process are depicted in Figure 2.2.

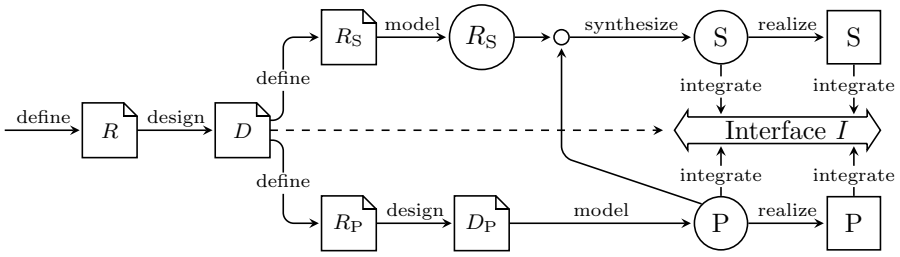

Figure 2.2: Systems engineering process with supervisory control synthesis [35].

Firstly the requirements R of the system under supervision are defined first and based on these requirements the design D of the system is created. These requirements are then decomposed in two subdivisions and the requirements R_p for the uncontrolled plant and R_s for the supervisor are specified. From the requirements of the plant, the design D_p is defined and a model P is created. On the other hand, in the case of the supervisor only the requirements are formally modeled. This results in a discrete-event model of the plant P and a formal model of the supervisor's

requirements R_s that can be used to synthesize a supervisor in the framework of supervisory control theory. These plant models can be used to analyse the behaviour of the uncontrolled plant under supervision also and the formal executable models built in the design phase can be used to assess correctness regarding functionality and performance of the designed components and the overall system.

In synthesis-based systems engineering, the required system features are used as input for the generation of the design of a supervisor that is correct by construction. As a consequence, verification can be eliminated from the analysis phase to a large extent. This changes the main development process from implementing and debugging the design and its implementation, to defining the system's requirements.

2.2 Behavioural models in MBSE

In this section, a brief presentation is given of all the models that can be used to define the dynamic behaviour of a component in the system in the model-based engineering approach which was presented in the last section. These models, also referred to as behavioural models, can be classified by their type of state, by the time domain of their dynamic behaviour and by the cause that makes their states change. With regard to the state of the system, the following types are possible:

- continuous state: The state is characterized by a combination of values that belong to a dense domain. A domain is dense if in between any two random values you can find another value.
- discrete state: The state is characterized by a combination of values from a discrete (also referred as countable) domain.
- hybrid state: The state is a combination of both continuous and discrete state components.

In terms of the time domain which describes the dynamic behaviour one can encounter:

- continuous time: the domain of time instants is dense, for instance the real numbers.
- discrete time: the domain of time instants is discrete, for instance the natural numbers.

At last, the main causes that make a state change are:

- time-driven: the state of the system changes on the basis of time flow.
- event-driven: the state of the system changes on the basis of events which refer to changes in the state of the system. The amount of time between a pair of consecutive events is normally not the same.

After the introduction of all the types of behavioural models one can now define a type of system called *hybrid systems*. These systems are used when modeling an uncontrolled plant in order to model physical quantities that change continuously over time and also modes of operation for different continuous dynamics that are based on sporadic inputs into the system or occurrence of combinations of state values. This leads to classifying these systems as both time-driven and event-driven.

Components like the supervisory controllers are most of the time modeled using discrete state even though continuous state and hybrid state are also possible and they are normally event-driven. The controlled plants of the system are of the same type as the uncontrolled plant. They are considered to be the composition of an uncontrolled plant and its supervisory controller. The requirements of the system are modelled as discrete state, time-free and event-driven systems.

They can also be modeled with the use of logical formulas but this falls out of the classification presented here.

The complexity of the models (in terms of their dynamics) influences the simulation in a direct way. Simulation is used for two main purposes: (1) to validate whether the uncontrolled plant is a reliable representation of the system and (2) to validate if the controlled plant behaves in the desired way.

2.3 Control strategies

As a general point of view, dynamic systems can be defined by a set of relations between its inputs and outputs so that for any given input this set of relations produces a certain output. Usually these relations are used to specify a specific purpose that the system needs to achieve from a control view. This means that the right input needs to be introduced in order to produce the desired output. To achieve this goal usually systems use a feedback loop resulting in what is referred to as a closed-loop system. This feedback loop introduces a control law that for a simple scalar case has the following expression:

$$u(t) = c(r(t), x(t), t)$$

In the above definition, $r(t)$ stands for the reference signal (also referred to as desired behaviour), $x(t)$ is the state of the system in any moment and $u(t)$ is the input signal that will be introduced to the system in order to achieve the reference. A representation in terms of a block diagram of this type of systems is shown in Figure 2.3.

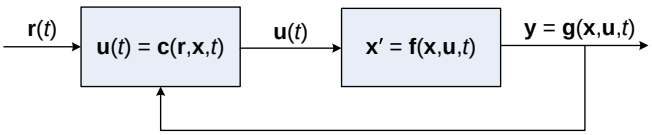

Figure 2.3: Closed-loop system [35].

This type of systems offer several advantages over the open-loop control system:

- The disturbances have less impact in the desired behaviour of the system.
- Errors in the parameters values of the models have also less impact in the desired behaviour of the system.
- Through the minimization of the error signal $(y(t) - r(t))$ one can continuously follow the desired reference automatically.

These are all cases of control strategies for time-driven systems but the truth is that the feedback control concept can also be used in discrete-event systems. In these type of systems, the plant of the system is modelled by a network of discrete-event automata, as opposed to the time-driven systems where a set of differential equations is used. This network represents all the possible sequences of events that can occur in the system. This is called the uncontrolled system behaviour. Normally this behaviour has to be restricted because it is not satisfactory in some ways and this leads to the introduction of a certain controller. A controller in the context of discrete-event systems is called supervisor and they must impose that the specifications are not violated. These specifications are expressed by automata or state-based expressions.

A graphical representation of the closed-loop control for a discrete-event system can be seen in Figure 2.4. In it, P is the uncontrolled system (also known as plant), S is the supervisory controller,

s is the sequence of events that are executed by P (and observed by S) and $S(s)$ is the set of events that are enabled and that P can perform in a certain state.

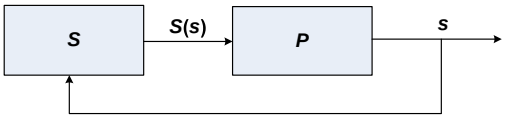

Figure 2.4: Closed-loop system for supervisory control [35].

2.4 Models for supervisory Control

As previously stated in Section 2 and schematically shown in Figure 2.1, the control of cyber-physical systems normally involves several layers of controllers. At the lowest layer you find feedback controllers which are based on the continuous representation of the systems components. At the highest layers you find the supervisory controllers which are represented as discrete-event systems that are suitable for dealing with situations like work cycle start-up and shut-down, change of operation modes, exception handling and recovery. Between these two layers there is an intermediate layer which contains an interface which is used as an abstraction of the system and low-level controllers for the purpose of the supervisory controllers. This way, information from the actuators and sensors is abstracted in the form of events while the commands from the supervisory controller are transformed accordingly to inputs signals to the actuators or set-points to the feedback controllers.

At the start of this section we explained the important role that models play in the development process and which different models can be used for certain purposes. A general scheme of the types of models, which are essential for supervisory control development, is presented in Figure 2.5 and explained as a workflow:

- At the start of the process the uncontrolled hybrid plant is developed in the form of several hybrid models of the physical components of the system. These models represent all the possible behaviours that the system can exhibit without any restrictions. Simulation and simulation-based visualization are the common tools to validate these initial models.

- From these initial hybrid models of the systems, an abstraction process can be executed in order to obtain the uncontrolled discrete-event plant possibly by using a hybrid observer that allows to abstract the information from sensors and feedback controllers in the form of events.

- The next step is the modeling process of the requirements. These requirements are related to the function that the system is desired to perform and along with the uncontrolled discrete-event plant, a supervisory controller can be synthesized.

- After that, the controlled system, which is the supervisory controller combined with the uncontrolled discrete-event plant, can be subject to an extensive verification (if some desired requirements cannot be expressed in terms of automata).

- Finally, by combining the supervisory controller and the hybrid observer, an observer-based supervisor can be derived whose main function is to translate command events to the appropriate inputs signals to the actuators or set-points to the feedback low-level controllers. As a final step, the actual real-time control code is generated, for the implementation of the controller in real-time.

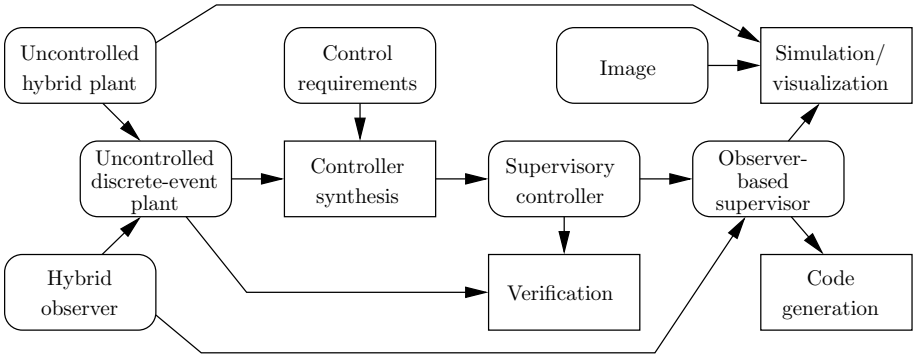

Figure 2.5: Scheme of models in the development process [35].

Chapter 3

Case study: Pick&Place Station of the xCPS platform

As previously stated, the system that will be used as a case study in this thesis will be the Pick&Place unit of the assembling area of the xCPS platform. This unit contains three actuators and five sensors in total and its main function is to pick up the top work pieces that come from a belt and drop them on top of the bottom work pieces that come from the index table.

The first actuator that is required to perform the function that this unit has is the picker that is used to pick up work pieces and drop them down, as can be seen in Figure 3.1. The state of this picker can be recognized by a sensor which indicates if a work piece is picked. This picker is connected to a robot arm whose motion is controlled by two actuators: a horizontal and a vertical moving robot arm. The inner and outer positions of the horizontal arm and the upper and lower positions of the vertical arm are both recognized by two sensors. In the initial position, the vertical arm is up, the horizontal is in and the picker has no piece picked. When a work piece comes from the belt next to the Pick&Place, the robot arm moves down so that the picker is able to pick it up. When the work piece is picked, the robot arm moves to its upper position and then the horizontal arm moves to the outer positions. Right after that the vertical arm moves down to its lower position and the picker drops the top work piece. Finally, after dropping the work piece, the robot arm moves back to its initial position using the reverse path.

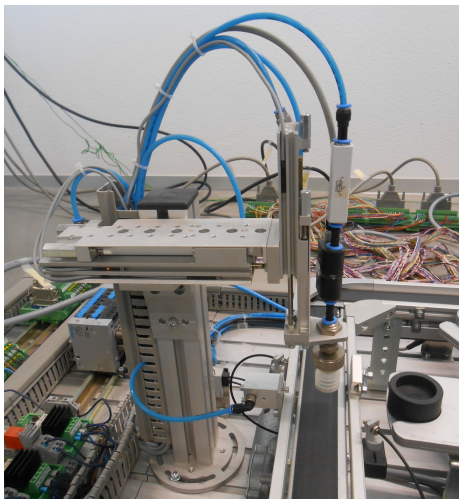

Figure 3.1: The Pick&Place station [23].

This Chapter is structured as follows. In Section 3.1 the CIF3 modeling language will be introduced since it is the only language for which models for the Pick&Place unit already exist [23]. In Section 3.2 there will be a lot of subsections dedicated to the modeling process of the whole uncontrolled system. This includes a full description of the uncontrolled hybrid plant in Section 3.2.1, a formal definition of the requirements of the system in Section 3.2.2 and the abstraction of the discrete-event plants in Section 3.2.3. Within Section 3.3, there will be an explanation of the synthesis process along the presentation of the obtained supervisor in Section 3.3.1 and a simulation of the controlled system in Section 3.3.2.

3.1 CIF3

The CIF3 toolset is an automata-based modeling language for the specification of discrete-event, timed, and hybrid systems [50]. This tool supports the whole development process stated in 2.4 including the specification of requirements and the uncontrolled plant, supervisory controller synthesis, simulation-based validation, verification, real-time testing, code generation among others. This is the main difference between CIF and other hybrid automata-based languages and tools that available nowadays (like Stateflow, MechatronicUML, HyVisual [33], SHIFT [19] or CHARON [2]) since these other tools do not cover the complete integrated tool chain for the development of supervisory controllers for complex CPS.

All the discrete models that will be presented in this section were created with the modeling language CIF3. Any physical component of the system that is being modeled can be modeled as an automaton, which consists of a set of different states that can be switch between the use of events. When making models for a large number of components, they can interact through a controller. For that to happen, the automata have to be defined as plants and within these plants the initial states have to defined for each component. In the models that will be presented in this section these two type of state specifications will be specified by an incoming arrow and a marked circle respectively. In addition to this, the events have to be specified as controllable or uncontrollable and in the latter case a dashed line will be used within the state models.

After defining the uncontrolled system, the requirements have to be specified by the means of automata in order to describe the behaviour of the events within each one of the plants. These requirements can be used in combination with the uncontrolled system to build a supervisor by applying data-based synthesis. This generates a supervisor that maintains the desired functionality for the system without letting it get into a deadlock.

Another important feature of the CIF3 toolset is its simulator. This simulator can be employed to validate each of the CIF3 models that have been mentioned before in isolation. In addition to this, it can also be used to validate the controllers when they are put in the context of the uncontrolled hybrid plant. This simulator allows for automatic testing of several different use cases, and the production of several forms of output.

3.2 Modeling the uncontrolled system

3.2.1 Uncontrolled Hybrid plant

As we saw in Section 2.4, the uncontrolled hybrid plant is used for simulation purposes along with the supervisory controller to evaluate the performance of the system and to see if improvements need to be made to ensure that the requirements are accomplished. This makes the model of the hybrid plant a very important model in the engineering process and the first one to be created during the design phase. The uncontrolled hybrid plant is a set of plant models in terms of hybrid automata that represent each one of the components that exist in the system.

This type of system has two distinct ways of state evolution. The first one is its progression over time within a location when certain continuous variables change their values continuously over time following a certain law or equation. The second one are the instantaneous transitions as a result of the occurrence of an event or events and the corresponding change of location and maybe of the values of the variables without any progress of time.

Therefore, the behaviour described by the hybrid automaton is a graph that consists of a certain number of different states (which represent the system states) and edges between these states that represent both progress in time and instantaneous transitions. The first are labelled by the evolution of the variables and the second one by an event or events. These types of systems are formally known as hybrid transition systems [15, 16].

Moving on to our particular system for the case study, as previously stated, the Pick&Place station has three actuators (named *Vertical_Motion_Ac*, *Horizontal_Motion_Ac* and *Picker_Ac*) and five sensors (named *Sensor_Vup*, *Sensor_Vdown*, *Sensor_Hin*, *Sensor_Hout*, *Sensor_Picker*). Each one of the actuators or sensors within the components contains one or more states and every one of these states is modeled as a location that can be switched by the firing of their transitions. In addition to this, some of these models contain continuous-time equations in some states that are used to represent the dynamic behaviour of some of the components (like the movement of the vertical/horizontal arm manipulator). This dynamic behaviour will be modeled with a set of dynamic variables that will be used in certain cases as the conditions guards for the transitions of the different components. These conditions will be transformed in events, as we will see in Section 3.2.3 when we do the abstraction of the hybrid plant to obtain the discrete-event plant. To complete the models of each components, the initial and marked states (which are needed for the synthesis process) are added just like it was explained in the previous section. As a result, we will obtain models per each component in terms of hybrid automata.

Before introducing any dynamic equation, several assumptions and clarifications have to be made about the system that is being modelled:

- The vertical and horizontal manipulators follow the same second-order differential equation (with the same parameters but with different variables and initial conditions) even though their real behaviour might be more complex. These differential equations will model the trajectories that each arm manipulator follows when they move.
- The picker performs its operations of picking/dropping the work pieces instantly and therefore there is no dynamic behaviour associated with it.
- The sensors by themselves do not have dynamic equations in their states but they do have control laws in terms of some dynamic variables as the guards of their transitions.

Therefore, the first things that need to be defined are the second-order differential equations used for the two arm manipulators. The equation looks as follows:

$$m\ddot{x} + b\dot{x} + kx = 0$$

Where x is the position of the robot arm, $\dot{x}$ is the velocity and $\ddot{x}$ is the acceleration in its movement. If this was the case of a body attached to a spring and a damper this equation would be Hook's law and the parameters m, b and k would be the mass of the body and the friction factors of the damper and the spring. In our case, the parameters lose their physical interpretation since the real dynamics are different from this law. Since we have two actuators (the vertical and the horizontal one) it is convenient to set two different dynamical variables. We are going to use the variable x in the horizontal movement and the variable y in the vertical movement.

The CIF3 toolset does not support the definition of second-order (or higher) differential equations to be solved by a certain solver that the tool provides. Instead, the differential equation has to be rewritten into several systems of first order equations and then, *continuous* variables and their derivatives (declared with operator *der* or $'$) can be used to implement the equations.

This second-order equation can be rewritten into a system of two first order derivatives by substituting v for $\dot{x}$ and $\dot{v}$ for $\ddot{x}$. The equation becomes (in the case of the horizontal movement):

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -\frac{1}{m}(bv + kx) \end{aligned}$$

and if you add initial conditions to the differential equations:

$$\begin{aligned} \dot{x} &= v \quad x(t_0) = x_0 \\ \dot{v} &= -\frac{1}{m}(bv + kx) \quad \dot{x}(t_0) = v_0 \end{aligned}$$

In our case study the dynamical parameters that have been used for the dynamic equations of the actuators (vertical and horizontal arm) are the following for both actuators:

- $m = 5$
- $b = 1$
- $k = 1$

What also needs to be added are the physical limits of the position variables that correspond to the trajectories of the two robot arm manipulators. In both cases it has been set that the trajectory is 10 cm long with the lowest limit set at 0 cm (inner and lower position) and the highest limit set at 10 cm (outer and upper position). After defining these limits we can state the initial conditions of the movements in all the possible cases:

	Lower→ Upper	Upper→ Lower	Inner→ Outer	Outer→ Inner
Position	$y_0 = 0$	$y_0 = 10$	$x_0 = 0$	$x_0 = 10$
Velocity	$\dot{y}_0 = 0$	$\dot{y}_0 = 0$	$\dot{x}_0 = 0$	$\dot{x}_0 = 0$

Table 3.1: Initial conditions of the different trajectories

As a result of this, now we can easily compute the trajectories of both manipulators and the value of their dynamic variables in any point in time. After defining the initial conditions, the only things that needs to be added into the models of the two arm manipulators is the events that allow them to change their states. Both manipulators start in a standing position (state *Standing_Still*) and they needs a certain event to start moving. For the vertical manipulator, two events are possible and this are the events *move_down* and *move_up* which will change his state to the *Moving_Down* or the state *Moving_Up*. The same holds for the horizontal manipulator with events *move_out* and *move_in* and states *Moving_Out* and *Moving_In*. On the other hand, a *stop* events is used in both manipulators to stop the movement and get back to a standing position.The models of this two manipulators can be seen in Figure 3.2 and 3.3.

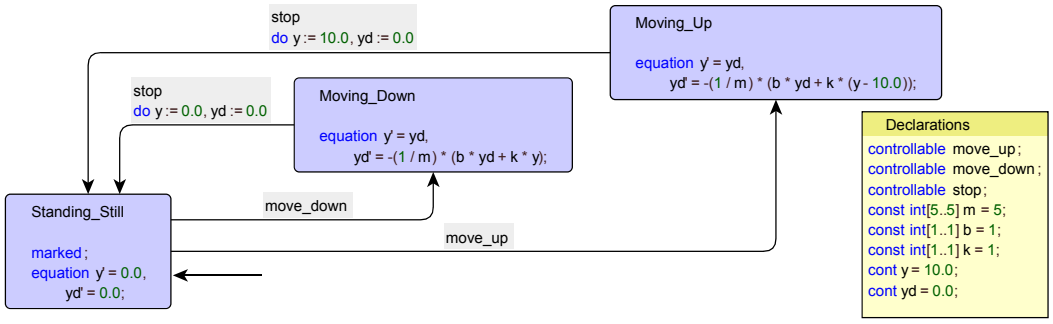

Figure 3.2: Hybrid model of the vertical arm manipulator.

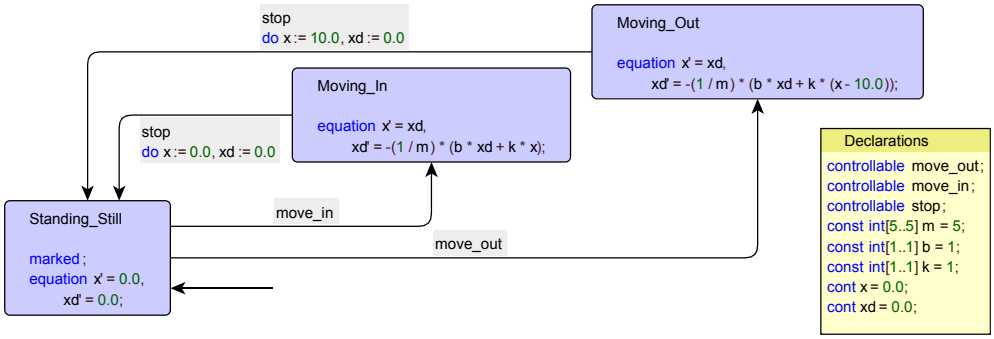

Figure 3.3: Hybrid model of the horizontal arm manipulator.

Moving on to the picker, since it does not contain any dynamic equations (due to the assumptions that have been made), the models can be easily created by defining the states that can have and the events that make them change. The picker has two states which are when it has a piece picked (state *Closed*) or when it does not (state *Open*). The events used to start/stop the picking process are the events *pick* and *drop*. The final model can be seen in Figure 3.4.

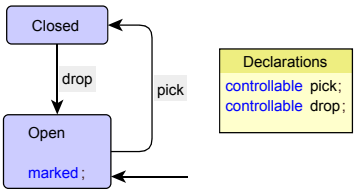

Figure 3.4: Hybrid model of the picker.

Once modelled the actuators, it is time to introduce the hybrid models of the sensors. To start with, there are two types of sensors in the xCPS platform: the sensors that are initially off and the ones that are initially on. The sensors that represent that the vertical arm is moving up (*Sensor_Vup*) and the horizontal arm is moving in (*Sensor_Hin*) are initially on. On the other hand, the sensors that represent that the vertical arm is moving down (*Sensor_Vdown*), the horizontal arm is moving out (*Sensor_Hout*) or that the picker has a work piece picked (*Sensor_Picker*) are initially off. Some of these sensors evolve their states because of a certain dynamic variable (in

our case the positions of the arm manipulators). The only sensor that doesn't is the sensors of the picker whose model can be seen in Figure 3.5.

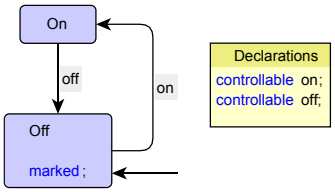

Figure 3.5: Hybrid model of the internal sensor of the picker.

On other hand, the other four sensors (the two of the vertical manipulator and the two of the horizontal manipulator) have conditions on the dynamic variables as guards for their transitions. These dynamic variables are the positions x and y of the two manipulators. As a result, we obtain 4 sensors that are controlled by position and with the following control laws that allow the events on or off of the sensors to happen:

$$\Rightarrow Sensor\_Vup.on \rightarrow Vertical\_Motion\_AC.y \geq y_{max}$$

$$\Rightarrow Sensor\_Vup.off \rightarrow Vertical\_Motion\_AC.y < y_{max}$$

$$\Rightarrow Sensor\_Vdown.on \rightarrow Vertical\_Motion\_AC.y \leq y_{min}$$

$$\Rightarrow Sensor\_Vdown.on \rightarrow Vertical\_Motion\_AC.y > y_{min}$$

$$\Rightarrow Sensor\_Hout.on \rightarrow Horizontal\_Motion\_AC.x \geq x_{max}$$

$$\Rightarrow Sensor\_Hout.off \rightarrow Horizontal\_Motion\_AC.x < x_{max}$$

$$\Rightarrow Sensor\_Hin.on \rightarrow Horizontal\_Motion\_AC.x > x_{min}$$

$$\Rightarrow Sensor\_Hin.on \rightarrow Horizontal\_Motion\_AC.x \leq x_{min}$$

With $y_{max} = 10$, $y_{min} = 0$, $x_{max} = 10$ and $x_{min} = 0$. With this, the models of the remaining sensors are finished and can be found in the Appendix A.3. As an example, the hybrid model of the internal sensor in the upper position can be seen in Figure 3.6.

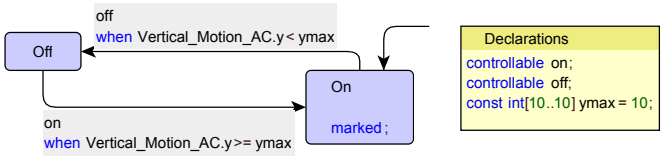

Figure 3.6: Hybrid model of the internal sensor in the upper position.

With this simple steps all the components of the uncontrolled hybrid plant have been created. Now it is time to describe the requirements that the system will have to fulfill to ensure the desired behaviour.

3.2.2 Requirements

These actuators and sensors that have just been described have to be connected with each other so that they work together as a unique unit (the Pick&Place station). This leads to the definition of the requirements for the system in the form of the order of the controllable events that can

be executed by the actuators. These requirements represent the path that the robot arm follows from its initial position until it drops a top work piece on a bottom work piece and comes back. One assumption that has been made in our system is that there will always be top and bottom work pieces available so in the end when we get the controlled system there will always be one event available to be performed. These requirements are shown in Figure 3.7.

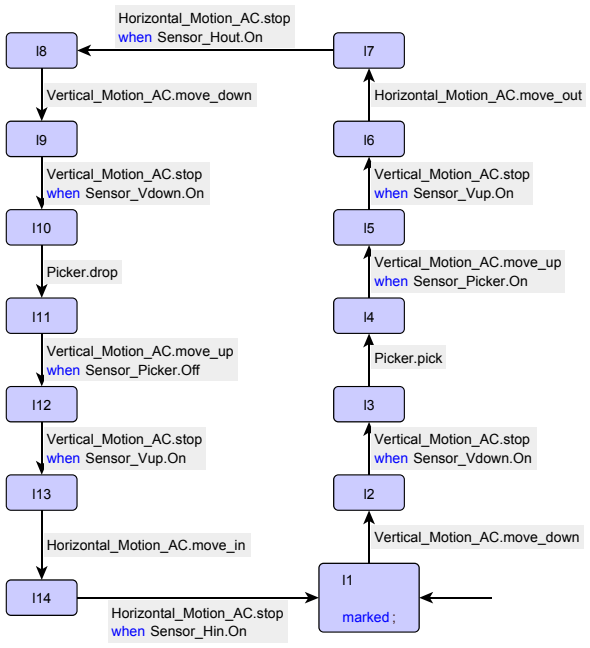

Figure 3.7: Definition model of the requirements for the Pick&Place.

In addition to these requirements regarding the path that the robot arm has to follow along time there are other requirements that need to be achieved. These requirements represent the interaction that happens between the different manipulators (vertical arm, horizontal arm and picker) and their respective sensors during the whole production cycle. These requirements are shown in Figure 3.8, Figure 3.9 and Figure 3.10:

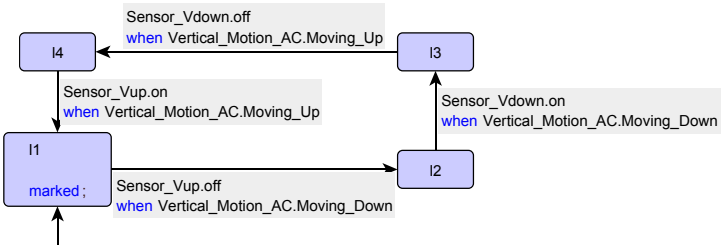

Figure 3.8: Definition model of the requirements for the internal sensors of the vertical arm.

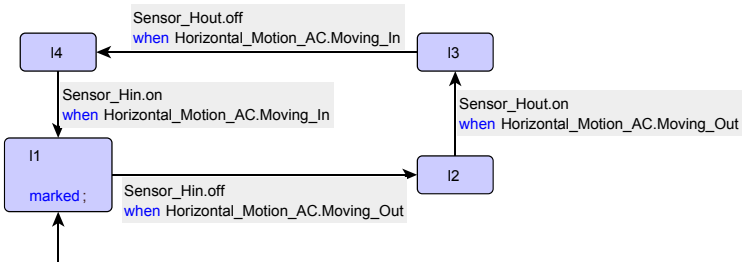

Figure 3.9: Definition model of the requirements for the internal sensors of the horizontal arm.

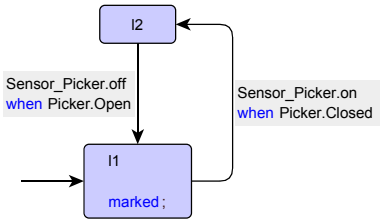

Figure 3.10: Definition model of the requirements for the internal sensors of the picker.

After having defined the requirements that the system needs to fulfill, it is time to model the uncontrolled discrete-event plant so that, along with the requirements, a supervisor can be synthesized after the synthesis process.

3.2.3 Discrete-event plant

After defining the requirements in the previous section, the only thing that is left is the uncontrolled discrete-event plant. This discrete-event plant is obtained by performing an abstraction of the physical behaviour of the actuators and sensors. As a result, we obtain models without any dynamic equations or variables neither inside the states nor in the transitions as guards. In the second case, these condition guards are replaced with the abstracted discrete events. An example of an abstracted discrete-event plant can be seen in Figure 3.11 where we can see the discrete-event plant of the vertical arm manipulator.

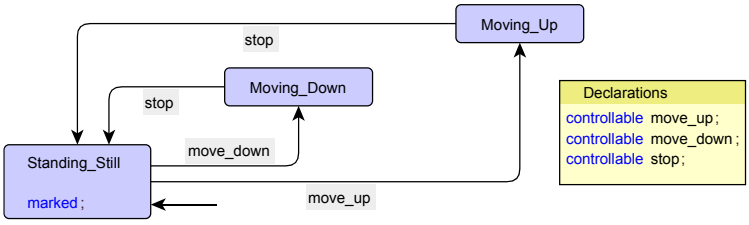

Figure 3.11: Discrete-event plant model of the vertical arm manipulator.

As one can see, the remaining automaton only contains the states and the transitions labelled with the events that enable them. The same holds for the discrete-event plant models of the sensors where the abstraction is performed in the transitions mainly. An example can be seen in Figure 3.12 where the model of the internal sensor of the upper position is shown.

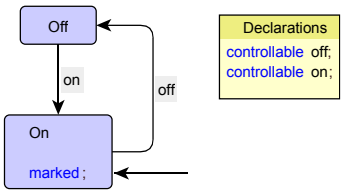

Figure 3.12: Plant model of the sensors that are initially on.

The rest of models can be found in Appendix A.1. Here we have joined the sensors depending on the initial state (on or off) to avoid repetition since some models are the same. Now let's move to the development of the controlled system which is explained in the next section.

3.3 Developing the controlled system

In this section we first focus on explaining the synthesis process that has been used in our case study in 3.3.1. In the same section, we will explain the behaviour of the supervisory controller that was obtained through the synthesis process and its main features. Finally, in 3.3.2 a simulation-based validation of all the models and the supervisory controller will be performed to ensure that the system has the desired behaviour at any time and that phenomena like deadlocks do not occur.

3.3.1 Synthesis process and supervisory controller development

As previously stated, Supervisory controller synthesis (or supervisor synthesis) is a method where one derives a supervisor (or supervisory controller) from a certain collection of plants and requirements models. The plant describes the capabilities of the physical system without any type of control strategy and the requirements model the functions that the system is supposed to perform (in other words, requirements restrict the behaviour of the plant). The goal of supervisory controller synthesis is to compute a supervisory controller that ensures the requirements, knowing the behavior of the plants, while preventing deadlock and livelock, and without restricting the system any further than is required.

In this thesis the synthesis algorithm that has been used is the data-based synthesis and we will describe it briefly. The basic data-based supervisory control problem is formulated has the following formulation. As it can be seen in [35], for a plant a supervisor is required such that:

- The controlled system is non-blocking. That is, it is possible to reach a marked state from all reachable states in the controlled system.
- The controlled system is controllable. That is, for all reachable states in the controlled system, the uncontrollable events that are enabled in the same state in the uncontrolled system are still possible in the controlled system. In other words, uncontrollable events are not restricted.
- The controlled system is maximally permissive (or simply maximal). That is, the controlled system permits all safe, controllable, and non-blocking behaviors.

This synthesis process can be done with the CIF3 toolset in two ways which are by either executing the data-based supervisor synthesis tool manually or with a script. On the other hand, there is an alternative process to the synthesis which is to build the supervisor manually through programming with the CIF3 language.

After the synthesis process, a supervisor should be obtained which might restrict the behaviour of the system so that no deadlock exists and the system is non-blocking. In our case, because of

the fact that all our events were controllable and the nature of the requirements and the plants the supervisor did not have to disable any events and the resulting controlled system already guaranteed the absence of deadlocks and the desired behaviour.

After having obtained the supervisory controller it is time now to validate its behaviour and see if the system performs as expected.

3.3.2 Simulation

The best way to validate that the supervisor you synthesized during the supervisor synthesis phase is correct, is through simulation. Simulation allows you to try different traces though the state space, to observe what happens for different scenarios (use cases) so we should try many traces and see the end-result. However in our case, because of all the requirements, there is only one possible trace of events so there is only one possible outcome and we have to make sure that this outcome is exactly the desired behaviour of the system.

To do this we will do the following things:

- Look at all the events in the trace and make sure that they satisfy the requirements.
- Check that there is always an event available to be performed (even if it is a time delay event when the manipulators are moving) and therefore there are no deadlocks.
- Watch the evolution of certain dynamic variables like the position of the actuators during time to make sure they relate to the physical reality that is happening in the Pick&Place station.

The first two were very easy check to by using the State Visualizer that you can enable when performing a simulation with CIF and they turned to be true. To validate the third one we used the Plot Visualizer to plot the values of the positions of the two robot arms and a variable named *pieces* that we added to the model that counts total the number of assembled pieces by the station. The results for a simulation time of 50 seconds are:

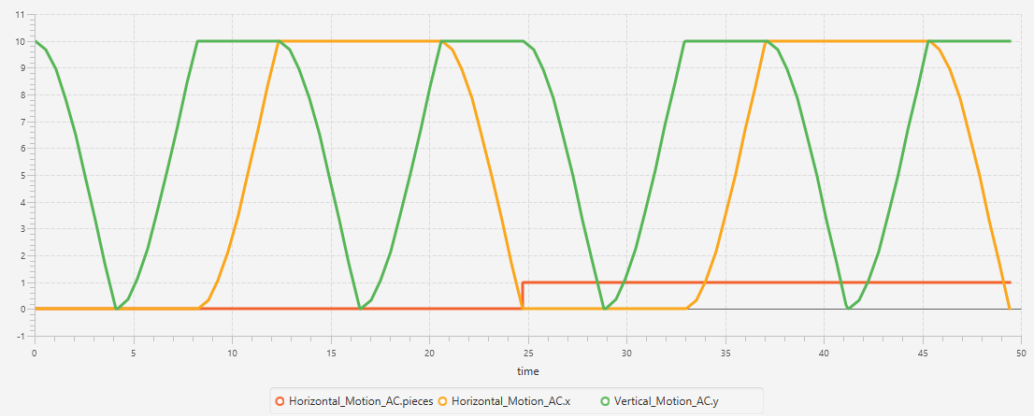

Figure 3.13: Evolution of the position of the actuators over time and total number of pieces produced.

As can be seen, the trajectories follow the second order differential law that was previously defined and after each cycle the value of the variable *pieces* is increased by one unit. Since this is a system with a very restricted behaviour because of the requirements there is not a lot more to do in order to validate it. One could do several tests changing the values of the dynamic variables but that would only change the trajectories that the manipulators would do and the ratio at which a piece is generated in the system. Therefore, we conclude that the validation process is over and that the controlled system has the expected desired behaviour.

Chapter 4

Stateflow & Simulink

Within the MATLAB environment developed by the MathWorks, one can find two interactive tools which are focused on model-based systems engineering. In its environment, MATLAB integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation [11].

Simulink [52] is a block diagram environment mainly focused on multi-domain simulation and model-based systems engineering which is used for modeling, simulating and analyzing dynamical systems. It provides a framework that allows to perform continuous testing and simulations to achieve the validation or verification of cyber-physical systems. In addition to this, some possibilities for code generation are provided. Simulink is mostly used to work with with linear/nonlinear time-driven systems and provides the user with a graphical editor, several block libraries for modeling different behaviours and solvers to support the modeling and simulating process. In addition to this, since it is integrated in MATLAB, this enables you to incorporate any MATLAB algorithm or function previously defined into the models and also to export the simulation results to MATLAB for a more extensive analysis.

Stateflow [53] is an environment whose main purpose is to allow he developer to create and analyze dynamic discrete-event systems based on finite state machines. It offers several possibilities to model your system in either a graphical or tabular representation and that includes: state charts, state transition diagrams/tables and truth tables. In the end, with Stateflow you can develop complex supervisory control structures to achieve the systems desired behaviour and that includes state machines animation and static and run-time checks for testing design consistency and completeness before implementation.

Stateflow is not an autonomous modeling tool, as mentioned before, since it is an extension of the MATLAB tool and operates in the Simulink environment. Simulink provides the developer with a graphical user interface for creating models as block diagrams, which can include a wide range of blocks, such as signals, different linear/nonlinear components and sinks. Within Simulink, also a Stateflow block (referred to as chart) is available so that a Stateflow chart can be added to a Simulink block diagram model. A Stateflow chart interfaces with the Simulink model. At the interface, physical connections between the chart and Simulink blocks and the exchange of data and/or events between the chart and external sources is defined.

In this chapter we will review the potential of these tools in terms of MBSE with supervisor synthesis capabilities. First, in Section 4.1 we will present the syntax and notation that Stateflow&Simulink use but focusing on Stateflow since that one is used for modeling and developing charts with supervisory control structures. Then, in Section 4.2 we will evaluate the capabilities of the Stateflow&Simulink toolset in terms of model-based systems engineering with supervisory control synthesis development. Finally, in Section 4.3 we will build the same case study that we did with CIF3 and will evaluate the simulations results and the modeling techniques that were used.

4.1 Basic modeling concepts

4.1.1 Simulink

Both Simulink and Stateflow are graphical languages and its graphical syntax is very intuitive. Any system modeled is described as a set of connected blocks (which communicate through the connectors attached to their ports) that compute several input/output relationships. For the purpose of modeling a system, Simulink provides with several different libraries of physical/dynamic components. The 6 most commonly used libraries are:

- **Continuous**, which contains blocks to process continuous signals and complex continuous time operators. This includes blocks like the *Derivative* and *Integrator* to derive or integrate a signal, or blocks like the *State-Space* and *Transfer Fcn* to model dynamical systems with state space equations or transfer functions in the s (Laplace) domain. Some of the blocks available in this library can be seen in Figure 4.1.

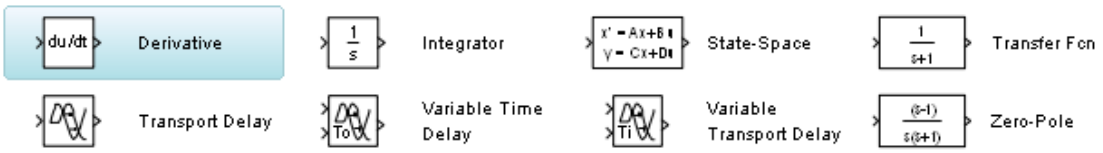

Figure 4.1: Examples of blocks of the Continuous library.

- **Discrete**, which contains blocks to process discrete signals and other discrete time operators. Most of them are types of transfer functions in the z (discrete-time equivalent of the Laplace domain) domain like the *Discrete Transfer Fcn*, the *Discrete Zero-Pole* or the *Discrete State-Space* blocks. Some of the blocks available in this library can be seen in Figure 4.2.

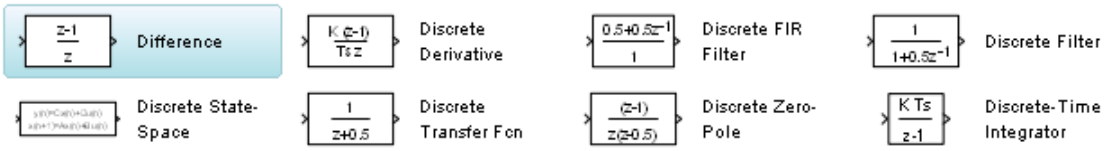

Figure 4.2: Examples of blocks of the Discrete library.

- **Math Operations**, which contains blocks for performing basic math operations on either variables or arrays. Some of these operations include the *Sum* (of the components of an array), *Add*, *Divide* or the *Abs* (absolute value of a variable) blocks. Some of the blocks available in this library can be seen in Figure 4.3.

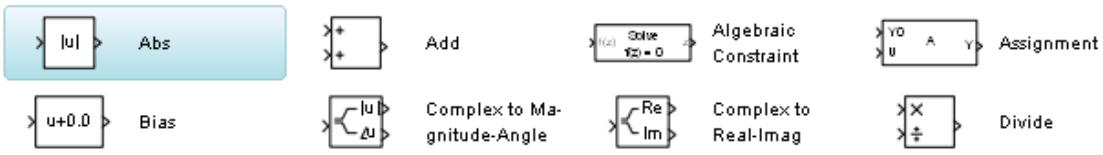

Figure 4.3: Examples of blocks of the Math Operations library.

- **Sinks**, which contains blocks for displaying and storing the results of simulations or to define boundaries of hierarchy. Some of the most common blocks include the *Scope* (platform to

plot the results), the *To File* (save results to a file) or the *To Workspace* (save results in the workspace) blocks. Some of the blocks available in this library can be seen in Figure 4.4.

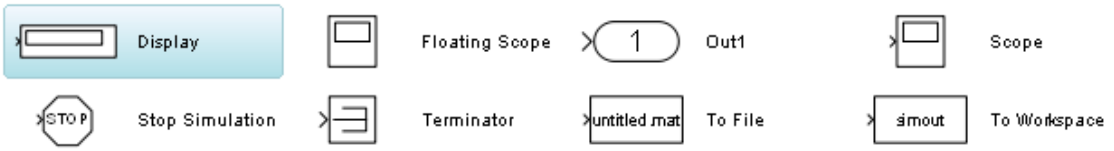

Figure 4.4: Examples of blocks of the Sinks library.

- **Sources**, which contains blocks to generate several different types of signals (usually by defining a function) that can be used as inputs in dynamic systems. Some of these possible singals include the *Step*, the *Constant*, the *Ramp*, the *Pulse Generator* or the *Signal Generator* block. Some of the blocks available in this library can be seen in Figure 4.5.

Figure 4.5: Examples of blocks of the Sources library.

- **Discontinuities**, which contains blocks for non-linear transformations of continuous/discrete signals. Some of these transformations might include the *Saturation*, the *Dead Zone* or the *Relay* blocks. Some of the blocks available in this library can be seen in Figure 4.6.

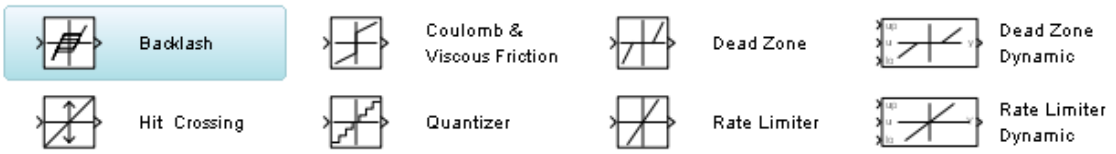

Figure 4.6: Examples of blocks of the Discontinuities library.

Combinations of these blocks (and the ones of other libraries) are used in Simulink to generate block diagrams, like the one in Figure 4.7, for modeling cyber-physical systems.

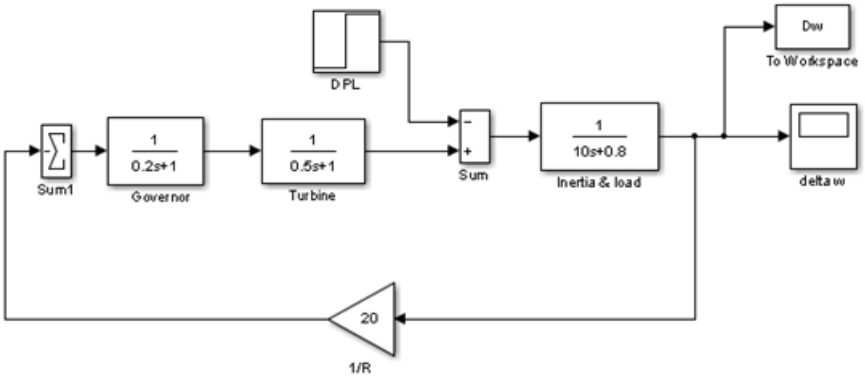

Figure 4.7: Example of a Simulink block diagram.

4.1.2 Stateflow

Stateflow uses mainly charts as a graphical representation of finite state machines (even though truth tables are also possible) when trying to represent event-driven (reactive) systems. For example, you can use Stateflow charts to control a physical plant in response to events such as temperature and pressure sensors, clocks, and user-driven events. A Stateflow Chart can use MATLAB or C as the action language to implement control logic.

A Stateflow chart usually contains a wide variety of graphical or nongraphical objects. Graphical objects include components like states, transitions, junctions, truth table functions or MATLAB functions and the nongraphical objects can include elements like events, messages or data objects. An example of a Stateflow chart with some of these elements could be the one that is shown in Figure 4.8.

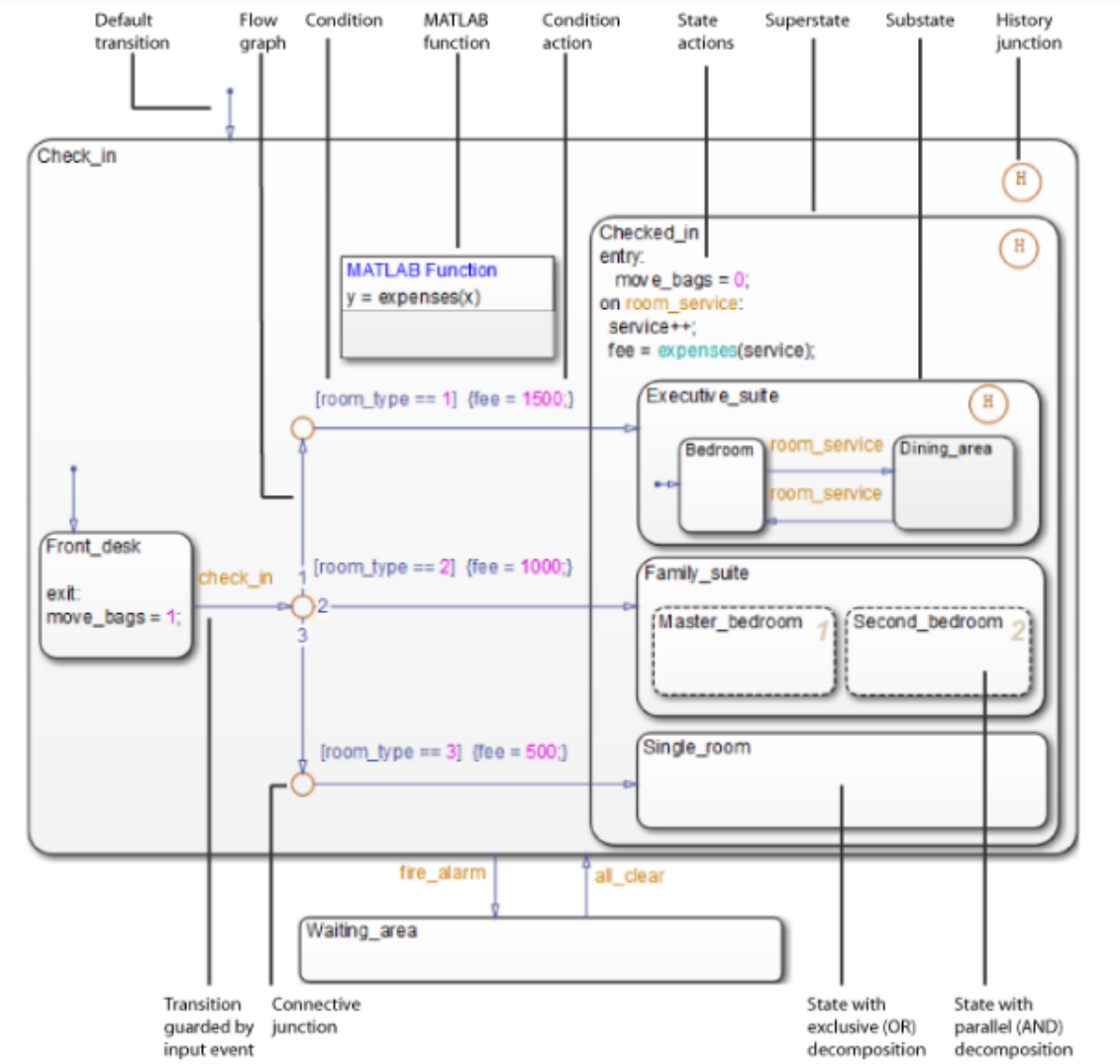

Figure 4.8: Example of a Stateflow chart [53].

States and transitions are basically the most important building blocks within Stateflow but objects like event, conditions, data objects, actions and junctions are also of major importance and will also be properly introduced.

States

A *state* describes an operational mode of the event-driven system that is being modeled. These states can either be active or inactive and this activity/inactivity changes depending on the events and conditions that occur during the execution of the state transition diagram.

In Stateflow, states are divided in either simple states or superstates and with the root state being the entire system. In addition to this, Stateflow allows to represent parallel processes by using AND-superstates and exclusive processes by using OR-superstates. The difference in their graphical representation is that the first ones are displayed with dashed lines for their contour and the second ones use solid lines. Within a state diagram chart, the different hierarchical levels that exist are defined by the boundaries of each one of the states.

The following is the general format for the label of a state which appears on the top left corner of that state [53]:

name/
entry: entry actions
during: during actions
exit: exit actions
on event_name: on event_name actions
on message_name: on message_name actions
bind: events

The following table explains the labels above and some of the most commonly used state action types:

State Action	Abbreviation	Description
entry	en	Executes when the state becomes active
exit	ex	Executes when the state is active and a transition out of the state occurs
during	du	Executes when the state is active and a specific event occurs
bind	none	Binds an event or data object so that only that state and its children can broadcast the event or change the data value
on event_name	none	Executes when the state is active and it receives a broadcast of event_name
on message_name	none	Executes when a message message_name is available.
on after(n, event_name)	none	Executes when the state is active and after it receives n broadcasts of event_name
on before(n, event_name)	none	Executes when the state is active and before it receives n broadcasts of event_name
on at(n, event_name)	none	Executes when the state is active and it receives exactly n broadcasts of event_name
on every(n, event_name)	none	Executes when the state is active and upon receipt of every n broadcasts of event_name

Table 4.1: State action types in Stateflow [53].

Transitions

A represent the change between two operational modes of the system and it is represented with an arrow that connects these two states. Its definition is defined by its label which contains: a trigger event, a condition, a condition action and a transition action. Its general format is [53]:

event_or_message trigger[condition]{condition_action}/{transition_action}

In Stateflow, any of these components in the transition label is optional. After being defined and during the system's execution, a transition can fire when its source state is active and the transition label is valid. This means that the trigger event must be enabled and the condition must evaluate to true in order for the transition to fire. There are the following types of valid transitions in Stateflow:

Transition Label	Is Valid If...
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Message only	That message occurs
Message and condition	That message occurs and the condition is true
Condition only	Any event occurs and the condition is true
Action only	Any event occurs
Not specified	Any event occurs

Table 4.2: Possible transitions in Stateflow [53].

In a Stateflow chart, transitions can not only be between states at the same hierarchical level. Transitions can occur between exclusive (OR) states, to and from junctions, between exclusive (OR) superstates, to and from substates and also self-loop transitions (from one state to itself).

Connective junctions

Connective junctions are used for the representation of different possible transition paths for a single transition and this enables the possibility to represent [53]:

- Variations of an if-then-else decision construct, by specifying conditions on some or all of the outgoing transitions from the connective junction
- A self-loop transition back to the source state if none of the outgoing transitions is valid
- Variations of a for loop construct, by having a self-loop transition from the connective junction back to itself
- Transitions from a common source to multiple destinations
- Transitions from multiple sources to a common destination
- Transitions from a source to a destination based on common events

Default transitions

A *default transition* specifies which exclusive (OR) state is entered when ambiguity exists among two or more exclusive states during the initialization of the chart. This type of transition has a destination state but no source state. Just like with the transitions between states, default transition can be labeled so that they can be activated by trigger events, messages, conditions and can perform actions. A default transition can also specify that a junction should be entered by default before entering any exclusive state in the chart.

Events

Events are non-graphical objects that are used to drive the Stateflow chart execution. With the occurrence of any event, the chart has to evaluate the status of its states and make the system

evolve if needed. As a result, an event broadcast can trigger a transition to occur and possibly the actions in the transition label and a change of the current state.

Events can be defined in different scopes and that determines where they are going to be used and which blocks are going to have access to its value:

Scope definition	Description
Local	Event that can occur anywhere in a Stateflow machine but is visible only in the parent object (and descendants of the parent).
Input from Simulink	Event that occurs in a Simulink block but is broadcast to a Stateflow chart.
Output to Simulink	Event that occurs in a Stateflow chart but is broadcast to a Simulink block

Table 4.3: Types of event definition [53].

Another thing to consider when defining events is the level of hierarchy that they are defined on since this will determine which states will have access to it and which will not. This information is summarized in the following table:

Where you define the event	To who the event is visible
Chart	The chart and all states and substates
Subchart	The subchart and all states and substates
State	The state and all substates

Table 4.4: Types of accessibility depending on where the event is defined [53].

Conditions

Conditions are boolean expressions that are contained in the labels of the transitions and that must evaluate to true for the transition to fire. They usually relate to the physical parameters of the system.

Data objects

Data objects are another sort of non-graphical objects which contain numerical values for their later use in a Stateflow chart. In terms of the scope of their definition, just like with events, data objects can either be local to the Stateflow chart, an input from an external Simulink block/system or an output to an external Simulink block/system. There are two basic types for data objects: constant when their value does not change during the execution time and parameter if their value changes due to a certain law or equation.

Condition actions

This type of condition is executed when the corresponding condition evaluates to true. This means that the action can occur even though the transition has not fired (since it may also require a trigger event for it to fire). Usually, actions are mathematical operations performed on data objects.

Transition actions

This type of actions is executed only if the transition fires. This includes the trigger event to be enabled and the condition to evaluate to true. Therefore, a transition action is performed when the destination state becomes active.

4.2 MBSE capabilities

4.2.1 Specification phase

In the specification phase, when trying to model the uncontrolled system, whether it is the hybrid or the discrete-event plant, the Simulink&Stateflow toolset does a great job in offering a lot of options and possibilities that allow the developers to model almost anything. From a practical point of view, Stateflow is a tool which is very easy to use and that has very few restrictions in terms of modeling (like languages with pre-defined model hierarchy) and this gives a lot of freedom to the end-user.

The time-driven low-level of the system can be easily modelled as block diagrams using some of the libraries that allow to represent continuous-time or discrete-time behaviour in terms of differential equations, transfer functions in the s or z domain or state blocks. Interactions between blocks can be created very easily by defining input/output ports per each block and then connect them using links (connectors) accordingly to the system's reality. To introduce input signals (steps, sinus, linear, etc) the source library offers a lot of blocks that can be easily used for that purpose. In case you want to implement some control strategies there are specific blocks for certain types of controllers (e.g. PID controller or a pole positioner) that are very intuitive and powerful for that matter. When trying to model some non-linear phenomena the discontinuities library usually has what is needed but if not there is always the possibility to use the MATLAB function block to program your desired functionality and add it to the diagram.

The low-level of the system can also be modeled with a Stateflow chart if the *Update Method* is set to *continuous-time*. Because of its nature, events cannot be defined in these types of charts but they do allow the definition of differential equations in the states of the automaton in a syntax very similar to the one used in CIF.

On the other hand, the developers define the discrete-event plants by using the capabilities offered by the Stateflow charts whose syntax was stated in the previous section. This chart allows to model the abstracted plant in terms of finite state machines (same formalism that CIF3 uses) and through all the objects that were also introduced previously almost all the functionalities that CIF3 offers can be achieved. Important concepts like mode switching (since states can have dynamic equations either continuous or discrete time) and update of variables (either by evolution of time or by changing states) are available but a concept like final states cannot be implemented. This is due to the fact that a Stateflow chart will always keep evolving its states if the right set of events is introduced and the data variables that trigger the conditions evolve accordingly.

Discrete variables are also available in these charts and its initialization is done manually when building the model. Variables (discrete, continuous or constant) can be shared in different ways. First of all, if they belong to the same chart they can be accessed by any state (in the same ways as in 4.3). Another way of sharing variables is by declaring them in the current workspace so they can be accessed by any block in the diagram (if they are not declared an error message will appear). Lastly, variables can be shared between blocks through the connections that are used to connect the different blocks of the diagram and that allow to share variables from Simulink blocks to Stateflow charts as inputs and variables from the Stateflow charts as outputs to the Simulink blocks as inputs.

Where Stateflow really deviates from CIF3 is in terms of synchronization. In Stateflow it is impossible to consider that multiple component models run simultaneously. At each time instant the parallel/exclusive (OR) component models run according to a certain order of execution that can be defined in the top right corner of each parallel component. This can develop in delays in terms of updating certain variables, switching modes between certain states or performing certain actions. There is also and issue in Stateflow when synchronizing events. In Stateflow there is not

the restriction that CIF imposes that an event can only happen if all the parallel states that share the event have to make a transition in that moment. Therefore in Stateflow you can have several transitions with the same event as trigger but they can perform that transition even if the other can't because they are in a different state. This problem has a solution though since there is a way to synchronize the events if wanted. This can be done with event broadcasting. You can broadcast events directly from one state to another to synchronize parallel (AND) states in the same chart. The following rules apply:

- The receiving state must be active during the event broadcast. Otherwise, the receiving state machine will not change its state for that broadcasted event.
- An action in one chart cannot broadcast events to states in another chart.

Event broadcasting is possible by using the Stateflow function *send* which has the following general format:

send(event_name,state_name)

where event_name is broadcast to state_name and any offspring of that state in the hierarchy. The event you send must be visible to both the sending state and the receiving state (state_name). The state_name argument can include a full hierarchy path to the state. For example, if the state A contains the state A1, send an event e to state A1 with the following broadcast:

send(e, A.A1)

Therefore, with event broadcasting it is possible to mimic the multi-party synchronization that we find in CIF if the right events are broadcast at the right state machines at the right time. This is for the developer to figure out when building the model.

If you want to simulate the behaviour of the uncontrolled plant there is no GUI input to select which event will happen at any given moment. In Stateflow there are two cases for generating events. The first one is when the events are declared as inputs. When the events are declared as inputs on the chart they have to be generated from external Simulink blocks or from other Stateflow charts that have those events declared as outputs. Events can be generated from Simulink by using the sources libraries to generate input signals with binary values. That is the common way to generate an input trace for the discrete-event plant to evolve in time. The other way of generating events is through broadcasting with the function *send* like we previously introduced. That is the case when the events are declared as local and then the function *send* can be used as an state action or transition action to generate events in a certain moment.

Those are more or less all the capabilities that Simulink&Stateflow have for modeling the plant models. On the other side, modeling requirements in terms of automata like in CIF3 is impossible in Stateflow and this affects directly the possibilities of having a synthesis algorithm to develop a supervisory controller. Requirements are modeled in Stateflow by adding or modifying the plant models. That might include changing the transitions, adding variables, adding events or adding actions in the states or the transitions. Since by doing this we are already building the supervisory controller we will explain it with more detail in the next subsection.

4.2.2 Supervisor development phase

As previously stated, there is no way to model the requirements in Stateflow so this affects directly the possibility of having an algorithm for applying synthesis and developing a supervisory controller. In fact, Stateflow does not provide any algorithm that can implement data-based or event-based synthesis but that does not mean that there is no way to build the resulting controlled system.

Since the supervisory controller is the one that restricts the plant's behaviour to make sure the requirements are fulfilled then the way to go in Stateflow is to modify the plants by adding all

the necessary of restrictions to implement the requirements and as a result you get the controlled system. In other words, a state chart in the end is the desired behavior of the discrete-event system, wherein sequences of events are controlled (by), conditions are used to specify requirements and actions can be used as control actions. A state chart by itself is a discrete-event controller. Thus, when building a state chart in Stateflow one in fact is constructing a supervisory controller.

For a discrete-event controller as a Stateflow state chart, the control actions are the execution of actions, mostly updating variables. These actions are guarded by conditions, which are manually derived from the requirements. Other control actions can be broadcasting events when certain variables reach certain values to model the abstraction process that CIF3 does between the low-level and the high-level control structures. Another control actions can be generating output events to send to other blocks in the Simulink diagram that might send back other events or that will change the dynamics of the system.

To sum up, the development of the supervisory controller and the controlled system is a manual process that can change for any different system and that does not have common guidelines to follow. Despite this, with all the capabilities that Stateflow offers and how intuitive the language is, it is not a tough process to develop a chart with the plant and the supervisory controller in it.

4.2.3 Validation phase

Once the supervisor is implemented in Stateflow as a state chart, the subsequent step is to validate the supervisor with respect to test cases by performing simulation-based visualizations in Simulink. When simulating in order to validate a design we usually look at two things:

- The evolution of the states of the system to see if they fulfill the requirements and that systems does not get into deadlocks when introducing a trace of events.
- The evolution of some variables of the system that might have some physical or technical interpretations. They are usually compared with some reference values that they should reach in a certain point in time.

Starting with the second one, any variable or parameter that is defined in a Simulink diagram or in a Stateflow chart can be plotted and analyzed. This includes any variable that is computed in a Simulink block (has to be defined as output), in a Stateflow chart (has to be declared as output) or any other variable that is defined through some mathematical operations in some point in the block diagram. The only thing that has to be done is to draw an arrow from the point where variable is created/computed and bring it to a Scope block from the sinks library to display it graphically.

Addressing the first one, Stateflow displays the active states from a chart (by changing the contour of the state to a blue color) at any time instant during the simulation so you can check if the states evolute accordingly to the requirements. You can also add breakpoints in state or conditions to make the simulation stop when they are reached and there is a step by step simulation that stops at any time that there is a change in the chart. This way you can check if the behaviour is correct after introducing a trace of events of the user's choice (with the procedure explained in the previous subsection).

Other features that might be interesting about Simulink include choosing the solvers that are used for solving the equations of the system, choosing the step-time of the simulation or integrating custom C/C++ code for simulation. To conclude, Simulink has enough tools to validate the controlled system and the supervisory controller in a graphical and intuitive way.

On the other hand, the Simulink&Stateflow toolset does not provide any methods nor possibilities for formal verification. Several transformations of Stateflow diagrams to input languages of various

formal verification tools have been created, but sometimes they do not support desired features of Stateflow or cannot be obtained. Two of the most relevant transformations that do not fall in the latter category are the ones in [13] and [54]. Therefore, proper formal verification methods for Simulink&Stateflow that support all the functionalities of the models have still to be properly developed.

4.2.4 Implementation phase

The Simulink&Stateflow toolset by itself does not offer any possibilities in terms of code generation to test the controllers in other platforms. To do that, you would need an extra license to acquire the Simulink Coder or the Embedded Coder. If you have a Simulink Coder or Embedded Coder license, you can generate C or C++ code from models that include Simulink diagrams and Stateflow blocks. Simulink Coder software generates source code that you can use for real-time and non real-time applications, including simulation acceleration, rapid prototyping, and hardware-in-the-loop testing. Embedded Coder software generates readable, compact, and fast code to use on embedded processors, on-target rapid prototyping boards, and microprocessors used in mass production. The generated C/C++ code does not contain code to interface with other blocks in a Simulink model or the MATLAB base workspace.

4.3 Case study

Following the first two phases that were mentioned in the previous section, the model of the Pick&Place station, along with the controller that ensures that the requirements are fulfilled, was developed and can be seen in Figure 4.9:

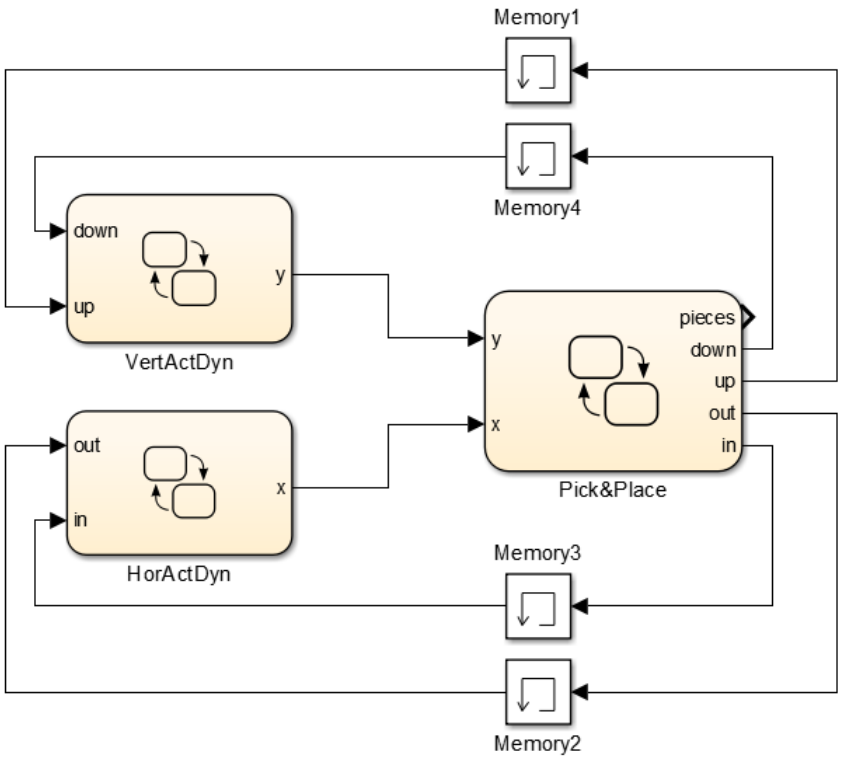

Figure 4.9: Stateflow model of the Pick&Place station.

As it can be seen, the model is formed by mainly 3 Stateflow Charts (*Pick&Place VertActDyn*

and *HorActDyn*), 4 memory blocks, and the connections that join them all together.

Pick&Place Chart

The Pick&Place chart (Figure 4.10) is the most important chart in the block diagram and the one that involves almost all the functionality of the system. This chart is divided in 8 automata that are used to model the 3 actuators (automaton *VertMotionAc*, *PickAc* and *HorMotionAc*) and the 5 sensors (automaton *SensUp*, *SensUp*, *SensPicker*, *SensIn* and *SensOut*). This chart uses local variables (*xmax*, *xmin*, *ymin*, *ymax*) and input variables generated in the other charts (x and y which represent the position of each actuator in the horizontal and vertical direction respectively) to model the transitions for the sensors of the two manipulators (always in terms of a boolean expression). The two manipulators also generate 4 variables (*up*, *down*, *in* and *out*) as outputs that are used by the other two automata to switch between states and model the dynamics of the manipulators correctly. One can see that there are a lot of events (*move_down*, *move_up*, *pick*, *drop*, *stop*, *on*, *off*, *move_in* and *move_out*) defined in this chart and they are all generated by event broadcasting with the command *send* that we introduced some sections ago. This chart also uses the variable *pieces* as a counter for the number of assembled pieces that are produced during the simulation. At last, the chart uses also a the variable *cont* as a decision factor to choose which events execute to fulfill the requirements that were presented in Section 3.2.2. The reason for that is that during the working cycle the vertical actuator goes up and down twice and *cont* counts the times that the manipulator goes down so then by doing the module of this variable divided by two the automaton can decide which is the correct path to take. This is modeled with a junction once the event *stop* is executed when the vertical manipulator is moving.

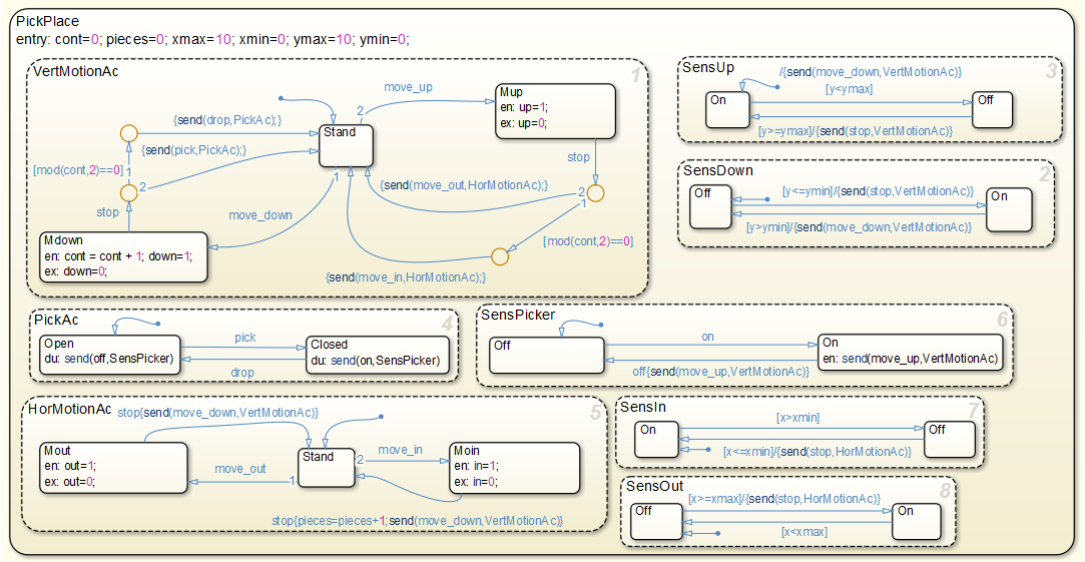

Figure 4.10: Stateflow model of the Pick&Place unit without the manipulators dynamics.

VertActDyn Automaton

The *VertActDyn* automaton (Figure 4.11) is a Stateflow chart that consists basically of 3 different states. These 3 states relate to the same states that the *VertMotionAct* from the Pick&Place chart has but here what is being modelled are the different dynamic equations to control the movement of the vertical manipulator. This can be done because the chart has the *Update Method* set to *continuous-time* so that by adding the sufix _dot one can model the derivatives of a certain variable (in this case the vertical position y and its velocity yd). The variable y that models the position is defined as output and is introduced as input to the Pick&Place station so that the vertical sensors

can change their states if that was necessary. The transitions between states are done through some conditions that involve boolean variables (variables *down* and *up*) that are generated in the Pick&Place and not with events because this types of charts (the ones with the update method as continuous time) do not allow events. The last thing that one can observe is that some variables are declared at the initialization of the chart and during some of the transition to guarantee the initial conditions of the trajectories that where defined in 3.1.

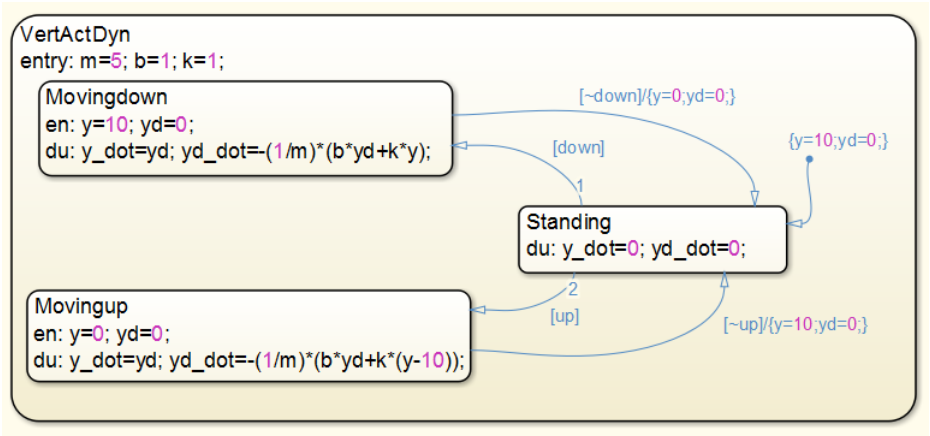

Figure 4.11: Stateflow models of the dynamics of the vertical manipulator.

HorActDyn Automaton

The *HorActDyn* automaton (Figure 4.12) is just like the vertical one but in the horizontal direction so there is nothing different about its behaviour or modeling strategy.

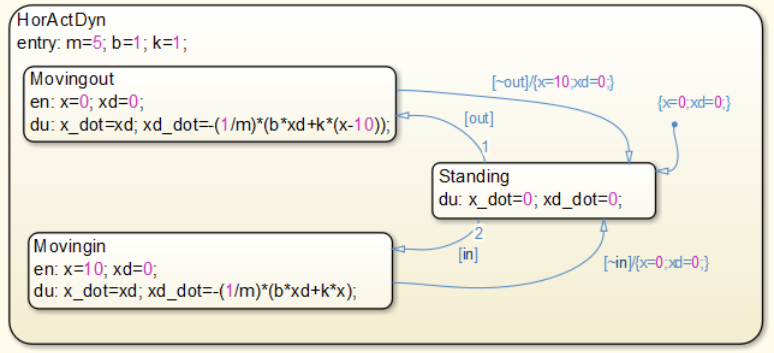

Figure 4.12: Stateflow model of the dynamics of the horizontal manipulator.

The last elements of the whole system that should be mentioned are the memory blocks. These blocks are used to connect the discrete variables that are generated in the Pick&Place chart to the two automata that run in continuous time and that model the dynamics of the manipulators. The memory block holds and delays its input by one major integration time step and it accepts discrete and continuous signals (in our case discrete variables).

Having already explained all the elements of the model it is time to perform the simulation and see if the results are similar to the ones that we have obtained with CIF. To do that we will plot the two positions (x and y) of the manipulators and the variable *pieces* that was also included in this model to count the number of assembled pieces. These results can be seen in Figure 4.13.

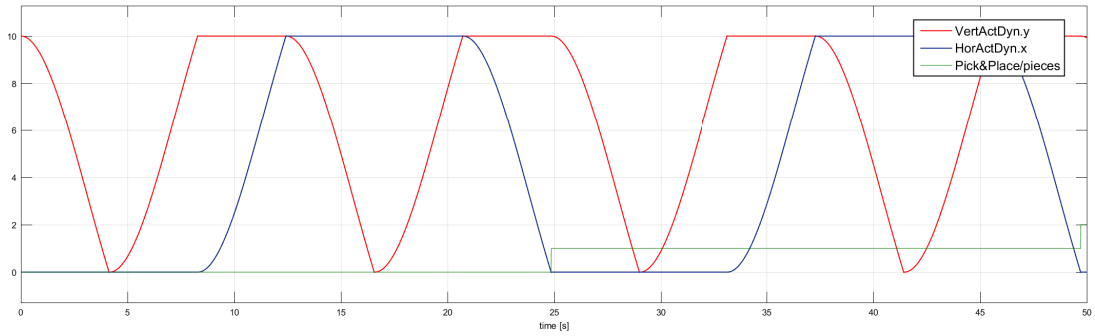

Figure 4.13: Results of the simulation of the model with Stateflow.

One can see than the results are almost (if not completely) the same as the ones previously obtained with CIF and that means that the modeling stage of the development process done with Simulink&Stateflow is correct. With this ends the comparison and evaluation of the Simulink&Stateflow toolset and now it is time to do the same with MechatronicUML in the next chapter.

Chapter 5

MechatronicUML

MechatronicUML [4] is a modeling language which uses concepts of the UML to provide a model-driven method for the development of the software embedded in mechatronic systems. In terms of modeling support, MechatronicUML supports the development of the structural as well as the behavioural aspects of almost any mechatronic system. This method follows a component-based approach [51] when it comes to software development and it specifically distinguishes component types as well as their instances.

The MechatronicUML method is focused on the software that is used for the real-time coordination of components in complex mechatronic systems. This coordination is basically achieved with the exchange of messages between systems components which results into complex discrete-event based behaviour in each one of the components. In addition to this, this coordination strategy has an impact on the dynamics of the physical components of the systems that are controlled by continuous-time controller software. As a consequence, MechatronicUML provides an integration process to connect the discrete-event based coordination behaviour with the continuous-time feedback controllers. MechatronicUML uses Real-Time Coordination Protocols (RTCP) to perform this real-time coordination.

In order to allow mechatronic systems to adapt to a changing environment and to coordinate and communicate with changing communication partners, MechatronicUML supports modeling the exchange of software components at run time. Besides the precise modeling of embedded software, another major aim of MechatronicUML is the formal verification of safety critical properties of mechatronic systems, which often operate in safety-critical contexts.

In this chapter we will review the potential of the MechatronicUML toolset in terms of MBSE with supervisor control development capabilities and will try to support it by trying to build the case study. First, in Section 5.1 we will introduce the basic modeling concepts that MechatronicUML uses when modeling complex cyber-physical systems. Then, in Section 5.2, an evaluation of the capabilities of the MechatronicUML toolset in terms of model-based systems engineering with supervisory control development will be performed. Eventually, in Section 5.3 we will attempt to build the same case study that we have been building with the other languages with an attempt to obtain the same results when simulating.

5.1 Basic modeling concepts

This section is partly based on [4].

5.1.1 Component Model

In MechatronicUML, the structure of the different elements of the system is specified by a component model. Any component represents a software entity that has an internal structure and

specific behaviour. That behaviour can be accessed by other entities through defined interfaces that show/expose this internal structure/behaviour through its ports. There are two types of components in MechatronicUML: atomic components and structured components.

Atomic Component

In MechatronicUML, atomic components represent the lowest level of abstraction of the component model. They contain different behaviour specifications which creates the distinction between continuous and discrete atomic components. On the one hand, the behaviour of continuous atomic components, which is defined with differential equations, cannot be specified in MechatronicUML and has to be specified with an external tool like Simulink. What MechatronicUML does provide is the specification of the interface of these components to share their information through their ports. On the other hand, the internal behaviour of discrete atomic components is specified in a Real-Time Statechart (RTSC). An example of these two types of components can be seen in Figure 5.1.

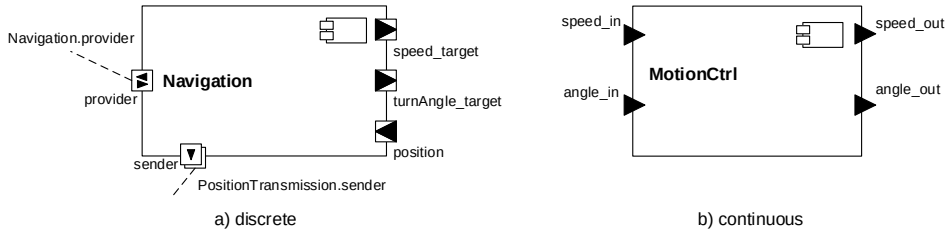

Figure 5.1: Concrete Syntax of Different Kinds of Atomic Components [4].

To define the interface of an atomic component, the developer has to specify a set of named ports so that run time the component's behaviour can be accessed. When it comes to ports, two types of distinctions can be made:

- Based on the direction
  - In-port (information flows into the port)
  - Out-port (information leaves the port)
  - In/out-port (combination of both)
- Based on the internal behaviour
  - Discrete port (discrete-event behaviour)
  - Continuous port (continuous-time behaviour)
  - Hybrid port (combination of both)

Discrete ports are used to connect discrete atomic components with a message-based communication protocol (known as Real-Time Coordination Protocol) for the correct exchange of messages during time. Continuous ports are used to connect continuous components by specifying a signal value and a data type. Finally, hybrid ports are used to connect continuous components with discrete components. The syntax of the different types of ports can be seen in Figure 5.2.

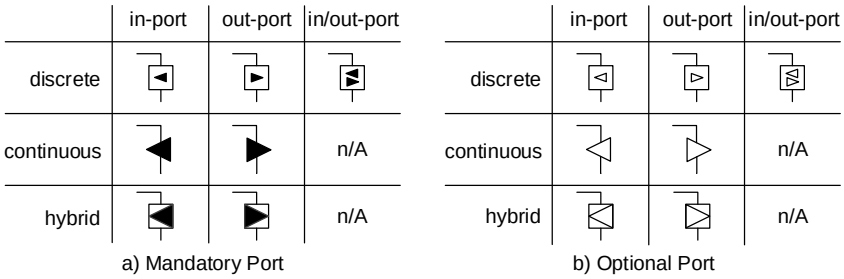

Figure 5.2: Concrete Syntax of Ports [4].

There is another distinction for ports that classifies them as optional or mandatory ports. If a port is optional then an instance of the port is not always active during runtime, i.e., it does not always send or receive values. If a port is mandatory, then an instance is always active during runtime.

Structured Component

A structured component is the result of embedding other components which can be either atomic or structured. This type of component does not require the specification of an internal behaviour since it is already specified by the embedded components. To create their interface, structured components define a set of ports just like it is done with atomic components but, in contrast with them, only continuous and discrete ports are supported. Just like with the structured components by itself, these ports do not need to have their behaviour specified. Instead, they are delegated to the port of the internal components by using delegation connectors so that the data type (for continuous components) and the set of send/received messages (for discrete components) can flow through the external interface.

5.1.2 Real-Time Statecharts

Real-Time Statecharts are models which are used to specify the behaviour of discrete atomic components or discrete ports. They are a combination of hierarchical UML state machines and timed automata [4]. Now we will introduce all the basic elements that are used in Real-Time Statecharts.

States

The basic element of Real-Time Statechart is the state which is used to represent a certain operational mode of the system. Apart from the normal states, per each Real-Time Statechart an initial state (where the system starts) has to be defined and many final states can also be defined if needed. When a final state is reached, no more operations are performed within the statechart. The template syntax of these two types of state can be seen in Figures 5.3 and 5.4.

Figure 5.3: Syntax Template of Initial State [4].

Figure 5.4: Syntax Template of Final State [4].

When it comes to the nature of the state, a distinction can be made between simple states, composite states, and orthogonal composite states. The first one has no hierarchy and no embedded components. On the other hand, a composite state can contain one or more regions which add a hierarchical level to the statechart and that contain a Real-Time Statechart inside. As a result, regions allow the developer to model orthogonal behaviour. In each region, one state is always

active if the parent composite state is active. An orthogonal composite state is a composite state with more than one regions inside. The general syntax that is used for simple states in MechatronicUML can be seen in Figure 5.5 along with an example in Figure 5.6.

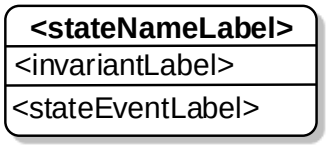

Figure 5.5: Syntax Template of a Simple State [4].

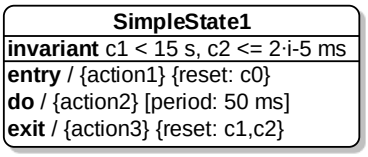

Figure 5.6: Example of a Simple State [4].

In order to avoid the system to stay too much time in a particular state, in MechatronicUML it is possible to specify time invariants to set the maximum time of stay in that particular state. This is defined with an upper time bound. If this upper time bound is reached before the outgoing transition fires, then the system will enter a deadlock.

Apart from invariants, the developer can also specify state actions. MechatronicUML supports these types of state actions:

- **Entry-Action**. The entry-action belongs to a state and is executed as soon as the state is activated.
- **Do-Action**: The do-action belongs to a state and is executed as soon as the execution of the entry-action is finished and has a period of execution specified in square brackets using a lower and an upper bound.
- **Exit-Action**: The exit-action belongs to a state and is executed as soon as the state is deactivated.

Actions are used to manipulate the values of certain variables or call operations that are allowed in the MechatronicUML action language. Other behaviour that can be implemented with an action is the following: resetting clocks, sending asynchronous messages and consuming asynchronous messages.

Transitions

Transitions are use to let the system switch between states. They are represented with and arrow that goes from the source state till the target state. These transitions can be labeled with features like guards, clock constraints, synchronizations, trigger and raise messages, transition-actions, and priorities.

The change from a source state to a target state happens when the transition is enabled (also referred to as transition fires). For that to happen, first of all, the guard and clock conditions must evaluate to true. In addition to this, if specified, must the asynchronous trigger message-event be available in the message-event pool and the synchronization channel be enabled. When the transitions fires, the exit-action of the source state is performed, the transition action is also performed, the entry-action of the target state is performed and the raise message-event of the transition is released [4]. Ultimately, a deadline can be added to the transition in the form of a lower and upper bound to specify the time interval where the transition must fire. An example of a transition in MechatronicUML can be seen in Figure 5.7.

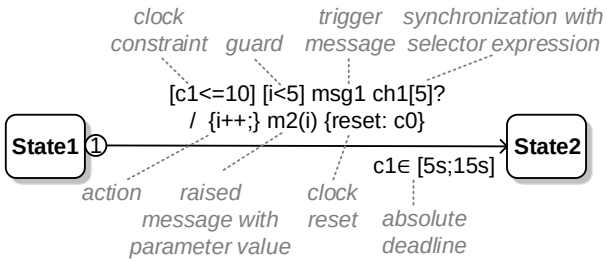

Figure 5.7: Syntax Example of a Transition [4].

Regions

As we previously introduced, the states of a Statechart can embed other Statecharts by using regions. A state can embed a set of regions while each region contains one Real-Time Statechart. That enables to specify hierarchical Real-Time Statecharts comparable to hierarchical UML state machines. The Real-Time Statecharts in all regions of a composite state are active if and only if a composite state is active. Each region has a name and if a state contains more than one region, each name of each region must be different.

Since regions embed Real-Time Statecharts, they may define own clocks as well as variables and operations. In addition, each region defines a priority that defines in which order the embedded Real-Time Statecharts of a composite state are executed. As for transitions, higher numbers indicate higher priorities.

Clock

In order to model time-dependent behaviour, Real-Time Statecharts use variables whose value increases continuously and synchronously at the same rate called clocks. Clocks are used for storing the elapsing of time during the systems execution. Every single clock within a statecharts must start at zero and must have a specific unique name. These clocks are mostly used in invariants, time guards, clock resets, and deadlines.

Constraints can be set on clocks in order to restrict the values of a particular clock to a certain range when defining a state invariant or restricting the enabling conditions of a transition. Clocks also allow reset in order to start running from zero again in some point during the execution of the system.

Variables and Operations

Another feature offered by MechatronicUML for the Real-Time Statecharts are the variables and the operations. A variable is a data-object with a unique name and specific value which can be restricted by the type of the variable (Real, integer, etc). The value of a variable can be changed by an action and it is also possible to define its initial value (even though it is not mandatory). Lastly, a variable can be set to constant so that its initial value is maintained during the execution of the system.

An operation is a callable behavioral feature that contains an implementation in a Real-Time Statechart [4]. Its definition is specified with its name, a set of input parameters and a return type. Each input parameter from the set is defined by its name and a data type and so is the return type. An operation can be called from the statechart that defines it and all the embedded components inside the same statechart. Like with actions, the specification of an operation is defined by an expression using the MechatronicUML action language.

Action Language

MechatronicUML defines an action language that enables the textual specification of imperative behavior. It supports the basic features of an imperative programming language including as-

signments, comparisons, boolean and arithmetic operations, loops, and if statements. This action language is used for defining transition guards and actions.

Asynchronous Message-Event

In MechatronicUML, messages are used to support the asynchronous communication between the interaction points of two discrete atomic components. A discrete interaction endpoint either corresponds to the role of a Real-Time Coordination Protocol (which will be introduced in the next section) or to a discrete port that refines a roles behavior [4]. Each one of these discrete interaction endpoints refers to a specific set of message types that might receive or send. In addition to this, message buffers can be declared to store the messages that the endpoint receives during execution time.

As we have seen, in MechatronicUML Real-Time Statecharts are used to model the behaviour of the interaction points between discrete components. This includes the supervision of the asynchronous messages that are send/received which are specified in the transitions of the Real-Time Statecharts as raise messages or trigger messages.

As we have seen, a trigger message is one of the requirements for a transition to fire. The transition will only fire if the specified trigger message of the specified type is available in the message buffer of the interaction point of the discrete atomic component. In addition to this, the transition that is going to fire can also get access to the message's parameters and these might be used to define the values for certain parameters of the transition's raise message.

On the other hand, raise messages are one of the possible effects when a transition fires. Therefore, when fired, a certain message of a specified type is sent through the associated interaction endpoint and transfer over the connector attached to it. Additional information can be send from the sender to the receiver by specifying several parameter types in the particular message type. These concrete parameter values can be accessed by the receiver once the transmission has been performed successfully.

Synchronizations

Synchronizations are used in MechatronicUML to allow two Real-Time Statecharts from two orthogonal composite states (embedded in their respective regions) to change their state at the same time. This leads to the possibility of defining a constraint that ensures that a transition does not fire alone, but along with another transition from another orthogonal statechart. We refer to this constraint as synchronization.

The first thing to do to define a synchronization is to specify the synchronization channel at an orthogonal state. This must be the parent state of orthogonal regions that contain the statecharts with the transitions that are going to be synchronized. After defined, the channel can be added to the transition label to specify the synchronization. Both transitions do not fire at the same time. The first one to fire is the sender transition and the second to fire is the receiver transition. Per each transition in a statechart only one synchronization channel (with one receiver and one sender) can be specified. There is another type of transition which includes a selector that is used to restrict the synchronization between two transitions from happening unless the evaluation of the selector's expressions are equal. This leads to the distinction of synchronization between plain synchronization (synchronizations without selector) and synchronization with selector. An example of the syntax of each of the two types of synchronizations can be seen in Figure 5.8 and 5.9.

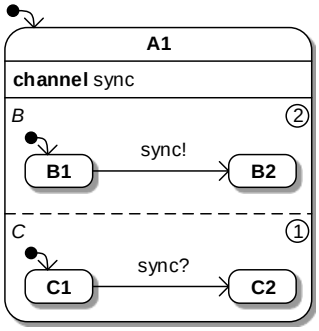

Figure 5.8: Syntax Example for a Plain Synchronization [4].

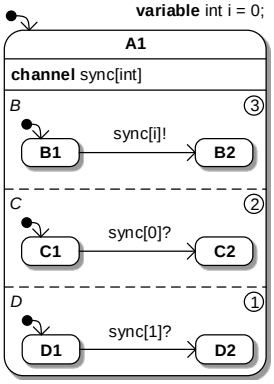

Figure 5.9: Syntax Example for Synchronization with Integer Selector [4].

A very important thing that should be noted is that MechatronicUML only allows one-to-one synchronizations and in particular no broadcast synchronizations. This means that in case of more than one sending and/or receiving transition, only pairs of one sender and one receiver transition are executed (which is chosen randomly).

5.2 MBSE capabilities

5.2.1 Specification phase

MechatronicUML follows a component-based approach for modeling the software that is embedded in high-tech cyber-physical systems. The modeling process is done with a visual component editor that provides the developers with components as building blocks for the construction of system architectures. From a practical point of view, the GUI that MechatronicUML offers is very intuitive to use, offering a certain amount of freedom when modeling and providing constant messages that allow the developer to see if he is doing something wrong.

As previously introduced, MechatronicUML is a modeling language for the model-driven development of the software embedded in high-tech cyber physical systems. This has a direct implication when trying to model the uncontrolled hybrid-plant of the system.

On the one hand, the time-driven part of the system (the low-level layer of the system) cannot be modeled in MechatronicUML. This results in developing the continuous/discrete-time plants and low-level controllers in a modeling tool like MATLAB/Simulink or Modelica/Dymola since they allow simulation by numerical integration. The only thing that MechatronicUML provides is the possibility of defining an atomic component as *continuous-time* in terms of its behaviour and to specify its ports and the type of variables that flow through those ports. The evolution of these variables or the dynamics of these components is developed in the same external tool.

On the other hand, MechatronicUML is a very powerful tool when modeling the event-driven part of the system (the high-level layer of the system). The behaviour of discrete components and their respective discrete ports is defined by Real-Time Statecharts (RTSC). These Statecharts are a combination of hierarchical UML state machines and timed automata so they model discrete-event systems with the automata formalism. Each component has exactly one RTSC that embeds the RTSCs of each one of the discrete ports in parallel regions. By the own nature of the Real-Time Statecharts, mode switching is guaranteed through the definition of the set of states and the set of transitions for each statechart that represent the discrete-event behaviour of a certain component.

RTSCs in MechatronicUML support event handling through the use of asynchronous message-events that are exchanged between the different ports of the different components in the system. Messages can be raise messages or trigger messages depending on if they are the result of firing a transition or a condition for the transition to fire. In each RTSC of the model, the developer has to specify an initial state and also has the possibility to define a final state where the statechart will end its execution (always regardless of any input event).

Continuous-time evolution is not supported in MechatronicUML and therefore updating variables in time following a certain dynamic law is not possible and neither is the synchronization on time since RTSCs do not implement Hybrid Automata. On the other hand, discrete variables can be declared in any RTSC and they can change their values by using the operations that are supported in the MechatronicUML action language. Variables do not need to be initialized before the execution of the system but it can be done if needed. Variables can be shared between statecharts if the ports of the statecharts share the variables through their connection. Finally, synchronization of events is supported in MechatronicUML through the use of plain synchronization and synchronization with selector depending on which one is required in a certain case. However, we have to be aware of the limitations that MechatronicUML has in terms of synchronization since they only allow one-to-one synchronization (one sender and one receiver) and only one synchronization channel per transition so broadcasting synchronization is not possible.

These are all the capabilities in terms of modeling the uncontrolled plant of the system. When trying to model the systems requirements the capabilities of the MechatronicUML toolset are very few. Just like with Simulink&Stateflow, MechatronicUML does not provide any possibility to model the requirements of the system in terms of finite state machines. This also leads to the absence of a synthesis algorithm for supervisory controller development but there is still a manual procedure to obtain the desired supervisor for the system. This procedure will be explained in the following section.

5.2.2 Supervisor development phase

Just like the other modeling languages/tools, MechatronicUML does not provide any sort of algorithm procedure to obtain a supervisory controller through event-based or data-based synthesis. However, this tool provides with a manual method that ensures the fulfillment of the requirements by using asynchronous communication and messages that represent events.

As we already know, the discrete interaction of components in MechatronicUMl takes place via its discrete ports that exchange asynchronous messages. Therefore, with the right coordination of the exchanged messages between the respective components the developer could obtain a control strategy that fulfills the desired requirements. To do that, MechatronicUML provides contracts that focus on the whole coordination via message-based communication that can guarantee hard real-time constraints and safety-critical aspects. These contracts are known as Real-Time Coordination Protocols (RTCP). Real-Time coordination Protocols define the behaviour of the coordination, defined using communication, which results in which message must be sent/received by which statechart at any given time during the execution of the system.

Therefore, the main objective of Real-time Coordination Protocols is to ensure correct message-based coordinations between discrete components. This results in every discrete port having their own contract/RTCP for the overall desired coordination. The MechatronicUML toolset provides with a visual editor, which is quite intuitive and easy to use, to perform this modeling task.

A Real-Time Coordination Protocol consists of the participants of the coordination (which we call them roles) and the role connector that defines the message transmission assumptions. Each role represents a discrete port. Within the graphical editor, due to the asynchronous communication, the developer has to specify the exchanges messages, the properties of the incoming message

buffers, and specific quality of the service assumptions of the RTCP like the message delay. On the other hand, in what the behaviour of the Real-time Coordination Protocols concerns, the developer has to model which role has to send/receive which message at which instant during execution time. To achieve this, Real-Time Statecharts have to be assigned to each one of the roles for them to be able to perform the correct message exchange communication.

Eventually, by using Real-Time Coordination Protocols, we can obtain a model where we have all our components with their respective ports connected, whose behaviour is described by RTSCs, connected with RTCPs to ensure that the end behaviour of the whole the system is the desired. This is the way that MechatronicUML has to obtain their supervisory controller. Once obtained, the process of validation is next and will be described in the following section.

5.2.3 Validation phase

MechatronicUML by itself doesn't provide a simulation environment to test the controllers in order to validate the behaviour of the modeled system. Therefore, in the case of MechatronicUML models, the component model and the corresponding Real-Time Statecharts must be transformed to the modeling formalism of an appropriate simulation tool (like Simulink or Modelica). Additionally, also the models that are developed in other disciplines such as the continuous-time controller or the continuous-time plant of the system must be integrated into the simulation model (unless the tool is the same). At the moment, MechatronicUML provides the translation of MechatronicUML models to MATLAB/Simulink and Stateflow and to Modelica.

The translation of MechatronicUML models to MATLAB/Simulink and Stateflow is properly defined in [28] and will be summarized briefly here. An overview of this transformation can be seen in Figure 5.10.

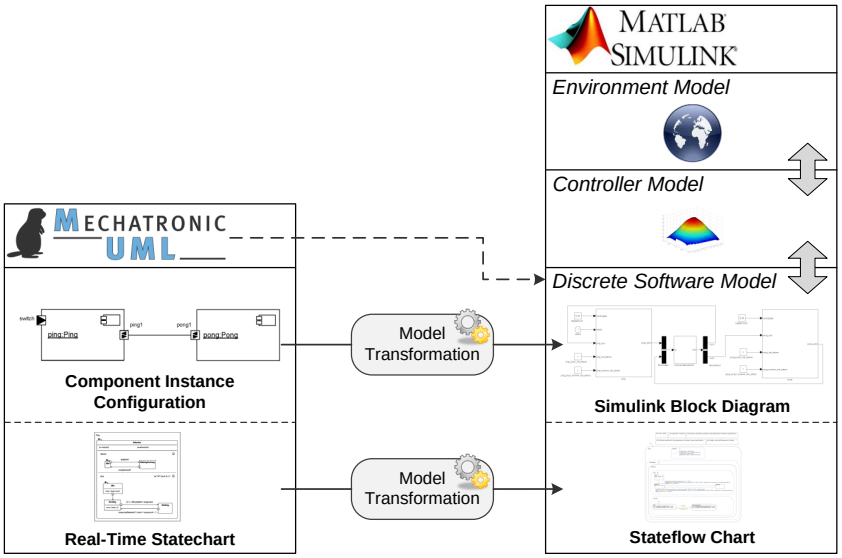

Figure 5.10: Overview of the MechatronicUML - Simulink&Stateflow transformation [28].

The transformation consists of two steps basically. In the first step, the model transformation transforms the component instance configuration of MechatronicUML into a Simulink block diagram. In this transformation, the assemblies of MechatronicUML are replaced by a communication switch implementation in Simulink which enables to deliver asynchronous messages from the sending component instance to the receiving component instance. In addition, the atomic components

are extended by the so-called link layer blocks that implement asynchronous communication with message buffers in Simulink.

In the second step of the transformation, the Real-Time Statecharts that define the behavior of the component instances are translated into Stateflow charts. The Stateflow charts are embedded into the Simulink subsystems that are created for atomic component instances. This includes the generation of helper functions for sending and receiving asynchronous messages which is not naturally supported by Stateflow.

The result of the transformation is a (hierarchical) Simulink block diagram with embedded Stateflow charts that specifies the discrete behavior of the mechatronic system. The interfaces of the controller model are contained in the MechatronicUML model in terms of continuous components. We generate corresponding subsystems for the controller model in our transformation such that the integration of the discrete software with the controller model is facilitated. The integration of the controller model with the environment model, however, needs to be carried out manually in Simulink by the developer.

For the translation of MechatronicUML models to Modelica there is no transformation procedure but a Modelica library that allows to build MechatronicUMl related models and use Real-Time coordination Protocols [56]. Once built, the models in the OpenModelica library, one can use its simulation features to validate the supervisory controllers developed.

The two transformations that we just mentioned provide a way of validating the controllers through simulation. However, MechatronicUML models and their Real-Time Coordination Protocols can be validated in another way in order to ensure that they fulfill the requirements that are needed for the safety of the whole system. This alternative for validation is by using model-checking [14]. This formal analysis technique checks that all the possible execution traces against the requirements. For each requirement of the accepted requirement language, the model checker is able to state if the requirement is fulfilled on all execution traces or not.

In MechatronicUML, real-time model checking of the Real-Time Coordination Protocols is supported via a model transformation to the model checker UPPAAL [5]. The procedure goes as follows: first there is a transformation from a MechatronicUML model to a UPPAAL model, then an automated execution of a model checking procedure is executed and finally a back-translation to a MechatronicUML is performed with all the details regarding the results. Therefore, the developer does not have to understand the UPPAAL language nor how the model checking procedure works because he only has to check the returning model that is the result of the back-translation. In the future it is expected that MechatronicUML will have its own model checking validation method.

Once validated the supervisory controller, either via simulation or model checking, the validation phase is over and it is time to see if MechatronicUML models can be implemented in certain platforms.

5.2.4 Implementation phase

The MechatronicUML approach, which consists of components and Real-Time Statecharts, allows to specify complex real-time systems following UML notations and the model-driven architecture (MDA) approach at the platform independent level (PIM). This platform independent description can then be mapped automatically to a platform specific model (PSM), provided that a target platform description in form of annotations describing real physical behavior is given [10]. Therefore, any programming language that supports real-time priority scheduling can be suitable for the implementation. Currently, MechatronicUML supports the code generation of Real-Time Java and C++ for an appropriate real-time operating system.

5.3 Case study

Since MechatronicUML does not support continuous-time modeling, the idea for the case study is to build the abstracted discrete event system resulting from the abstraction process of the uncontrolled hybrid plant. After building it, it would be interesting to explore one of the transformations that we talked about in section 5.2.3 in order to try to validate the controlled system. If we wanted to validate the models through simulation we would have to introduce the continuous-time part of the system that is missing in the MechatronicUML model in order to perform the complete simulation.

In order to model the Pick&Place station, the first thing to do would be to create the atomic components that represent the discrete-event system (actuators and sensors). The next thing would be to build the Real-Time Statecharts that represent the discrete behaviour of the atomic components and their respective ports. After that, Real-Time Coordination Protocols would be created in order for the different ports to exchange the messages during the execution.

In order to see how many Real-Time Coordination Protocols should be created let's take a look at the message-event exchange that each pair of components should have in order to fulfill the requirements. Per each pair of components that exchanges at least 1 message in one direction a Real-Time Coordination Protocol will be needed:

- *VertMotionAct* → *Sensor_Up*: events *on*, *off* / Sensor_Up → VertMotionAct: event stop
- *VertMotionAct* → *Sensor_Down*: events *on*, *off* / *Sensor_Down* → *VertMotionAct*: event *stop*
- *VertMotionAct* → *HorMotionAct*: events *move_in*, *move_out* / *HorMotionAct* → VertMotionAct: event *move_down*
- *VertMotionAct* → *Picker*: events *drop*, *pick*
- *HorMotionAct* → *Sensor_In*: events *on*, *off* / *Sensor_In* → *HorMotionAct*: events *stop*
- *HorMotionAct* → *Sensor_Out*: events *on*, *off* / *Sensor_Out* → *HorMotionAct*: event *stop*
- *Picker* → *Sensor_Picker*: events *on*, *off*
- *Sensor_Picker* → *VertMotionAct*: event *move_up*

As we can see, there are certain components that will need several Real-Time Coordination Protocols since they interact with more than one different component (in a unidireccional or bidireccional way). Therefore, certain components will have more than one discrete ports in order to allow all the needed protocols (for example, the *VertMotionAct* will need 5). What we have to remember also is that every single discrete port has to have a Real-Time Statechart that defines its behaviour and they cannot be the same since MechatronicUML does not tolerate that.

As a result, several Real-Time Statecharts with the same structure (states and transitions) will have to be build in order to represent the behaviour of each one of the discrete ports. Each one of these parts will have different trigger and raise messages since they are part of different Real-Time protocols and therefore exchange different messages with different statecharts. Finally, the final step towards getting the controlled system would be to synchronize all the statecharts of the same ports in order for them to evolve together in time.

Here is where the problem begins. As previously stated, MechatronicUML only supports one-to-one synchronization and therefore the discrete ports of components like the *VertMotionAct* would need more than one synchronization channel per transition in their Real-Time Statecharts for them to evolve together. As we know, that is also not possible since MechatronicUML restricts

the number of synchronization channels per transition to one and therefore the there is no way to synchronize all the transitions of the discrete ports of a component if that component has more than two ports.

This restrictions that MechatronicUML has in terms of event broadcasting suppose a major impediment to building the supervisory controller that implements the desired behaviour for the Pick&Place station. Therefore, no modeling results can be presented for the MechatronicUML in this particular case study and of course no simulation results can be presented either.

Chapter 6

Modelica

Modelica [25] is an object-oriented equation-based programming language commonly used to design, develop and analyze high-tech complex cyber-physical systems. This language is different from modeling tools like Simulink in certain features. Modelica's modeling method is based on equations instead of assignments and this allows for acausal modeling with equations that do not specify the data flow direction which causes a re-usability of the Modelica classes. Modelica also offers a multidomain modeling capability which results in models that contain several components from different physical domains like electrical, hydraulic, mechanical, biological and control and that are well connected and defined. In addition to this, Modelica is an object-oriented language with a general class concept that unifies classes, generics known as templates in C++ and general subtyping into a single language construct [25]. Finally, Modelica also offers several features/constructs for creating, describing and connecting many types of components which makes Modelica a great language for describing the architecture of complex cyber-physical systems.

From the developer's point of view, models in Modelica are basically described by object diagrams that contain mainly components and their connections. A component usually represents a physical element (like a resistor in a electric circuit) which has ports (in the case of the resistor it would be the two pins) that allow for interactions (in the case of the resistor it could be an electrical flux) with other components through connectors. By joining all the components all together the developer obtains the physical system that he wants to study in the form of an object diagram model. It is worth to note that each one of the components has a specific internal behaviour which is defined in a bottom level by a set of equations written with the Modelica syntax.

To help the developer during the modeling process, Modelica structures the different types of components regarding their nature (electrical, hydraulic, etc) in different libraries (also referred to as packages) which make them more accessible when modeling. It is worth remembering that Modelica is only a language, and therefore a proper simulation environment is needed to graphically edit and define a Modelica model and to perform different simulations and analysis techniques to analyze its behaviour.

In this thesis the implementation environment for Modelica models has been OpenModelica [26]. This environment consists of several interconnected subsystems, as depicted in Figure 6.1.

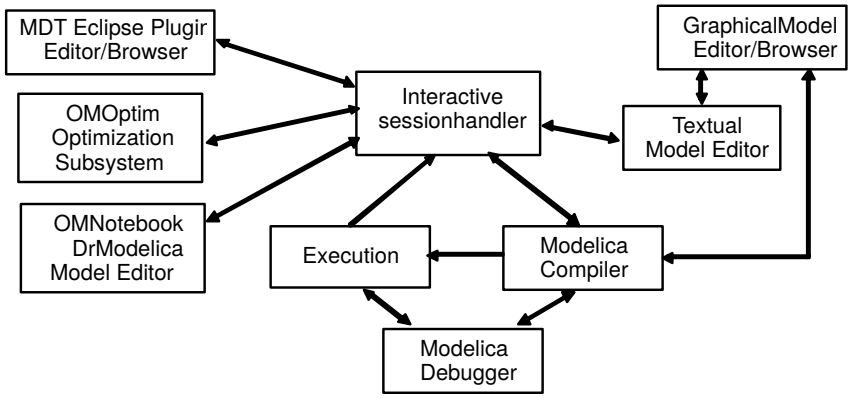

Figure 6.1: The architecture of the OpenModelica environment. Arrows denote data and control flow [26].

In this chapter we will review the potential of the Modelica language in terms of MBSE with supervisor synthesis capabilities and we will try to support it by trying to build the case study. First, in Section 6.1, a description of the basic syntactic statements and modeling concepts that the Modelica language uses in the context of hybrid systems with supervisory control structures will be provided. It is not a complete reference but only a selection of the basic constructs of the language. Then, in Section 6.2, an evaluation of the capabilities of the Modelica language in terms of model-based systems engineering with supervisory control development will be performed. Eventually, in Section 6.3, we will attempt to build the same case study that we have been building with some of the other languages/tools with an attempt to obtain the same (or similar) results when simulating.

6.1 Basic modeling concepts

This section is partly based on [25].

Modelica is a typed language for hierarchical physical systems that programmers that work with Java or C++ will find a lot of similarities with. Modelica provides with some primitive data types like String, Boolean, Integer, Real and Enumeration and, just like in C++ and Java, it is possible to build more complicated data types through the definition of classes. Classes are the basic construct of Modelica and they define how an object works. This modeling language provides with many types of classes: models, blocks, functions, packages, types and records.

Modelica also provides control statements and loops. The two basic control statements are the *if* and *when* statements and the two basic loop statements are the *while* and *for* statements. Only the syntax of the *if* statement will be presented since it is the only one that has been used in this thesis:

if expression **then**
  equation/algorithm
elseif expression **then**
  equation/algorithm
else
  equation/algorithm
end if

When attempting to describe complex hierarchical system architectures, Modelica uses its powerful component model with components that can be connected through the Modelica connection mechanism and that can be visualized as connection diagrams. This framework focuses on components and its connections and makes sure that the communication works over the different connections.

In order to add certain functionalities to the cyber-physical systems designed with Modelica, there are some libraries that should be mentioned. The first one is the *EmbeddedSystems* library which contains special communication block implementations that allow simulation of non-ideal communication between controllers and plants. Some of the features that can be simulated are: sampling, computational delay (fraction of sample period), communication delay, noise measurement and signal limitations. The second one is the *LinearSystems* which contains several functions that provide with different representations of linear, time-invariant differential and difference equation systems along with several typical blocks to define different control loops and strategies. Finally, the last library to be mentioned is the *Synchronous* library which extends the scope of the Modelica language to allow the modeling of synchronous control systems with variable sampling rates and by enabling an automatic code generation for embedded systems.

6.1.1 State Machines

Originally, Modelica was intended as a language primarily used for physical systems modeling but with Modelica 3.3 the scope has been extended to complete systems that include state machines in their behaviour [20]. In Modelica, a state of a state machine is any block that does not have continuous-time algorithms or equations in it. A cluster of block instances at the same hierarchical level which are coupled by transition equations constitutes a state machine. The transitions between these blocks contain conditions (also referred to as guards) that allow the transitions to fire. State machines are modelled in Modelica with the state machine extension which is a substitute for the *StateGrap* library [39] that was used before.

Before introducing the basic syntactic elements that are used in Modelica for building state machines, an example of a hierarchical state machine created with Modelica will we introduced an explained to show the scope of capabilities that Modelica offers. We will talk capabilities but we will not mention the specific syntactic declarations that are used for them, that will come later. This example can be seen in Figure 6.2:

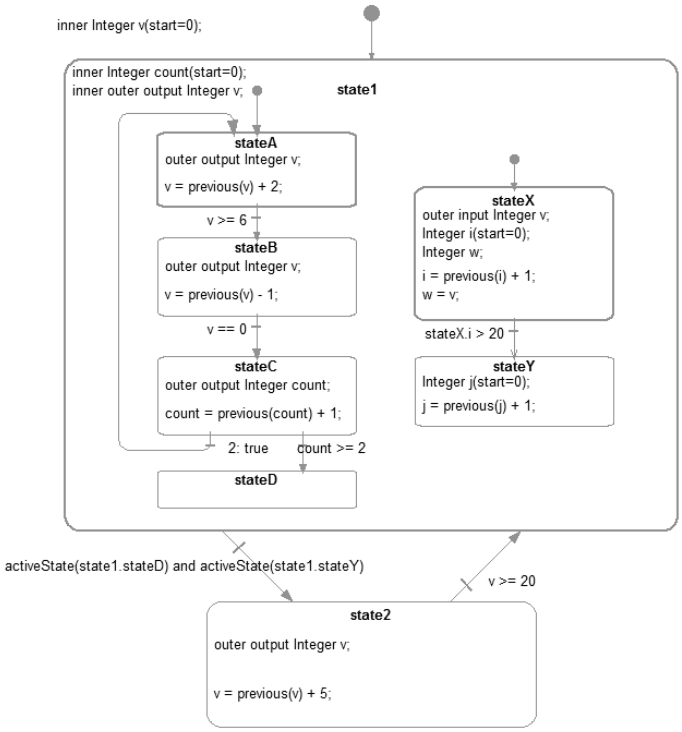

Figure 6.2: Example of Hierarchical State Machine in Modelica 3.3 [20].

The model of the state machine demonstrates some features that should be noted. First of all, with the Modelica state machine extensions, there is the possibility of building hierarchical state machines with orthogonal states with their basic elements like states and transitions. Therefore, there are meta states (like *state*1) with one or more state machines in it working in parallel. As it can be seen, states are instances of ordinary blocks with data-flow equations and the transitions between such blocks are represented by connections associated with transition conditions. Even though it cannot be seen, the transitions can be immediate (strong) or delayed (weak) when it comes to them firing. Another thing that cannot be seen but it is very important is that all parts of the state machine have the same clock and, therefore, the variables of active/inactive states can be accessed/read whenever the clock ticks. Finally, the possibility of sharing variables through the different levels of hierarchy is supported in Modelica with some syntactic declarations that will be introduced later.

Defining a state machine

When creating state machines in Modelica, not many declarations have to be used in order to obtain the desired functionality. The main declarations are: parameter definition, state definition, transition definition, sharing variables, and some other useful declarations.

The first declaration that is usually done is the definition of the global variables/parameters that will be (probably) accessed by the states machines. These are defined with a syntax like **(parameter) type name(start=value)** where **parameter** is optional (if not used it is considered a variable with a value that changes following a certain law), **type** refers to the type of the variable (**Real**, **Boolean**, etc) with name **name** and starting value **value**. This declarations can also be made in local states if needed.

Next, the definition of any state is done with the following declaration:

block NameState
  <parameters definition>
equation
  <statements>
end NameState;

Where NameState is the name of the state, the parameter definition is done just like we have introduced and in the **equation** compartment, the dynamic equation are typed and the state definition is always encapsulated with the **block** NameState **end** NameState; clause. Following state definition, the definition of transitions is also quite easy and it has a single declaration for it which is:

transition(from,to,condition,immediate,reset,synchronize,priority)

Where **from** and **to** are the source and target states, **condition** is a boolean condition that must evaluate to true for the transition to fire, **immediate** is a parameter that if evaluates to true it creates an immediate transition but if evaluates to false it creates a delayed transition, **reset** is a boolean variable that if evaluates is true it resets the current state machine and the values of all its variables, **synchronize** is used to disable the firing of the transition unless all the other state machines have reached a final state (state without outgoing transitions) and **priority** is used to set the different priorities when a state has more than one outgoing transitions.

Variables are shared among different hierarchical levels by using the **inner/outer input/output** notation. The **outer** declaration is used in states to access a variable that is defined in the above hierarchical level and that is defined as **inner** in that level. In Figure 6.2, *stateA* accesses the value of the variable v with the **outer** declaration for example. On the other hand, the **input** declaration is used when a variable which is accessed with an **outer** declaration and is going to be used in the current state but will not change its value (it might part of an equation/transition).

This is the opposed case than the **output** declaration following an **outer** declaration which is used to access a variable of a parent state and change its value in the process. When modeling intermediate level states one will use a combined declaration like **inner outer output** which is the only case where the **inner/outer** are combined. Also, the **inner** declaration is never followed by **input/output** unless it is in an intermediate state.

The last useful declarations that are used in Modelica are the declaration **initialState(state)** which is used in each state machine to set the initial state and **activeState(state)** which is used to know if a state from a certain state machine is active at the moment where the clock ticks. A last thing that should be noted is that the **transition**, **initialState** equations can only be used in **equation** compartments and cannot be used in *if*-equations with non-parametric conditions or in *when*-equations.

Modelica extensions for state machines

As previously stated, any block without continuous-time equations or algorithms can be a state of a state machine in Modelica 3.3 [20]. This restriction eliminates the possibility of having Hybrid Automata in Modelica. However, an extension for the Modelica 3.3 state machines is presented in [21] to overcome this through multi-mode modeling. The basic idea for multi-mode modeling is to extend the Modelica 3.3 synchronous state machines to continuous-time state machines that have continuous-time models as states. No new Modelica language element is needed, only a generalized semantics for state machines has to be introduced. The concepts have been evaluated with a Dymola prototype.

To sum up, a proposal is presented in [21] for modeling variable structure systems with dynamically changing number of states in Modelica by extending the synchronous clocked state machines to continuous-time state machines. With this extension it is straightforward to model hybrid automata. However, hybrid automata are not practical to use for physical system modeling. A novel extension is proposed to use continuous-time acausal models as states of a state machine. By mapping connections to connectors on a state machine in a particular way on equations, the standard symbolic processing for Modelica models can be applied. This approach allows already handling a large class of useful variable structure systems with dynamically changing sizes of continuous-time states.

Models cannot be handled with this new method, if connections between state and non-state components lead to constraints on continuous-time state variables that vary for the different state machine states.

The proposal is not yet complete. Especially, mappings for all connector types of Modelica need to be still defined. The proposed extension is also still not supported in terms of simulation in platforms like OpenModelica (since the prototype was tested in Dymola) and therefore this makes it reach less people that program with the Modelica language.

6.2 MBSE capabilities

6.2.1 Specification phase

Modelica is a typed object-oriented modeling language oriented towards complex cyber-physical systems which are modeled in a graphical equation-based approach. Within this approach, each graphical icon represents a physical component like an hydraulic pump, a DC motor or a spring. These icons are connected with composition lines that represent the actual physical connections like a heat flow or an electrical line. The variables in the interfaces of these icons describe the

interaction between the different components whose behaviour is described with equations. Because of these features and more, Modelica becomes a very graphical and intuitive language for modeling and simulating (with a simulation environment) high-tech systems.

When it comes to modeling the uncontrolled system, Modelica offers several interesting features. The time-driven part of the system can be modeled with component diagrams that can mix components from multiple disciplines like electrical, mechanical, hydraulic or thermal. This component diagram approach deviates from a block diagram approach like in Simulink in the way that is way more intuitive to see how the system looks and what you are modeling. Control strategies can be easily implemented with the *Controller* library that was previously introduced and that provides with different control strategies depending on the needs of the developer. Both continuous-time and discrete-time operating modes are supported in all the components and in all the features regarding the control strategies to be applied. As a whole, the low-level of the system can be fully modeled with a high degree of detail with all the capabilities that Modelica provides.

On the other hand, the high-level part of the system (the discrete-event plant abstracted from the uncontrolled hybrid plant) can be modelled by using the state machine extension that Modelica 3.3 provides. Therefore, Modelica supports the automata concept (the timed one) and with the extension [21] the hybrid version of the automata is also supported. As a result, basic components such as states, transitions and actions can be implemented with Modelica. When it comes to the states, one has to define the initial state of each one of the state machines in the system but the concept of final states, which tools like MechatronicUML offer, cannot be defined. The transitions between these states can be easily created with a simple function that offers features (apart from the source state, destination state and the condition) like the priority of the transition or the synchronization parameter.

However, when comparing Modelica with tools like CIF or Stateflow, the first feature in which Modelica is lacking is when modeling events. In Modelica you cannot model a system in terms of automata in which the transitions are guarded by the events that result from the abstraction process. Modelica only supports the events directly from the low-level part of the system, i.e. when the variables reach certain limits that make the system evolve in a certain way. Another way to represent events is by defining boolean variables that are used as conditions for the transitions and that they change their value in a certain state of the state machine. As previously stated, mode switching is supported (even though with certain limitations) thanks to the extension [21] that is still under development.

The state machines in Modelica can also have variables which can have a continuous-time or discrete-time evolution, depending on the equations that are used for each one. Therefore, the variables can be updated upon time in any state of the system. It has also been seen that thanks to the *inner* and *outer output* notation it is possible to share variables among different levels of hierarchy of the state machines of the system as long as the variables are well declared in the states where they are used. Every single variable has to be assigned an initial value.

Synchronization in time is provided but the synchronization of events is not and it should not be confused with the *synchronize* parameter that can be used when modeling transitions. This parameter is used to disable that transition until all the state machines within the from-state have reached their final states and not to make two or or more transitions evolve together at the same moment.

To conclude, even though it has some limitations, the Modelica language still offers several possibilities when trying to model the uncontrolled hybrid plant of the system. On the other side, just like in Simulink&Stateflow, there are no possibilities regarding modeling the requirements in terms of automata or any other formalism. Therefore, this will directly affect the possibilities of having an algorithm for the synthesis process and obtain a supervisory controller through this process.

6.2.2 Supervisor development phase

The fact that the Modelica language does not offer any possibilities in terms of modeling the requirements, just like the other modeling tools, affects directly the possibility of having a synthesis algorithm for supervisory controller development. However, there is still a possible manual process to obtain the desired supervisor even though it may have some limitations.

The manual process for obtaining the supervisory controller in Modelica resembles a lot to the one used with Stateflow but with less possibilities. In Modelica, the supervisor needed for the plant to have the desired behaviour is basically built by addressing the transitions between states, and in particular their conditions. The conditions of each one of the transitions are built to represent the events of the low-level system and in a way that they fulfill the requirements. One difference with Stateflow is that in Modelica there are no actions to be performed when a transition is fired and neither when a state is entered or is left. Actions are only performed during the whole period in which a state is active and that represents a limitation and also a change in functionality in comparison with Stateflow for example. Since event broadcasting is not supported, a way to represent events that allow to fire transitions is to create boolean variables used as the conditions of the transitions and change their value when it is needed.

Overall, the manual process for obtaining a supervisor is quite simple but limited at the same time and this results that for very complex high-tech systems it is almost impossible to obtain a supervisor that is able to supervise the uncontrolled hybrid plant since a lot of features that are needed (event broadcasting or actions for example) are not implemented or are lacking.

6.2.3 Validation phase

Since Modelica is a modeling language and not a tool, the validation phase is closely conditioned on the simulation environment that is chosen to perform the model simulations and analysis. Two of the most popular simulation environments used for Modelica models are OpenModelica and Dymola and the first one is the one that has been used in this thesis.

The OpenModelica environment has several subsystems (as can be seen in Figure 6.1) and its main parts are:

- **OMEdit**: this is the graphical model editor/browser that OpenModelica uses for component model design by connecting the instances of each Modelica class and browsing Modelica model libraries for picking components models. It also includes a textual editor to manually edit any model and a window for interactive command evaluation.
- **Interactive session handler**: this is used to type and interpret commands and expressions from Modelica to evaluate, simulate, plot among other operations. It also contains a simple history facility and completion of file names and certain identifiers in commands.
- **OMOptim**: this is an optimization subsystem for OpenModelica used for design optimization allowing to choose an optimal set of design parameters for a model. It provides an intuitive graphical user interface to work with.
- **OMC**: this is the advanced interactive OpenModelica compiler which is used mainly to compile Modelica code to C for simulation. There are also other language platforms to generate code to. This subsystem also includes facilities to build simulation executables.
- **Execution and run-time module**: this subsystem is used to execute/run the compiled code (which comes from the translated functions and expressions) and the simulation code (which comes from the equation-based models linked with the numerical solvers that are going to be used).

With all these subsystems, OpenModelica provides a friendly and intuitive environment that allows a lot of plotting possibilities that can be used when trying to validate if the behaviour of the system under the control of the supervisor is the desired one. However, we have to remember that this will be used for complex cyber-physical systems but still with some limitations or assumptions since the supervisory controller will probably not be able to restrict the systems behaviour strictly as desired. But for all the cases when the supervisor is built completely, OpenModelica offers enough tools to validate its performance.

On the other, neither Modelica nor OpenModelica offer any possibilities in terms of formal verification of the requirements of the models. The reason behind this reality is that there are certain Modelica practices that directly make Modelica unable to support formal methods. These Modelica practices are stated with detail in [31]. Some of these problematic features are: compiler interaction, artifacts for numeric simulation, unnecessary component model complexity, algorithms, sequential states, and incomplete models.

6.2.4 Implementation phase

In OpenModelica, as it has been seen, there is a subsystem called the OMC compiler which is the one in charge for the code generation. The OpenModelica compiler has a series of options in terms of its execution which are referred to as flags. One of these flags is the flag *-simCodeTarget* that is used to set the target language for the code generation. The valid options as far as target language is concerned are:

- C (default option)
- CSharp
- C++
- Adevs
- sfmi
- XML
- Java
- Javascript

Therefore, OpenModelica offers a lot of possibilities when it comes to generating code to implement in a certain platform. From an overall point of view, the code generation process done with OpenModelica is a very simple and intuitive process and with the additional help that the platform offers even more.

6.3 Case study

To build the Pick&Place station of the xCPS platform in Modelica, the code in Appendix C has been used. This code models the system and works in the following way:

- The model has 8 state machines in total, just like in the other languages. The state machines have the same states than in the other languages with the exception of the one that models the vertical manipulator. Just like in Stateflow, a variable *count* was used to allow the automata to know at which part of the production cycle was at and, because of the nature of the actions in Modelica (they execute during the whole time that the state is active) a state had to be added between the states *StandingStill* and *MovingDown* to allow this variable to increase 1 unit. This state, named *Intermediate*, has the same dynamic equations than the state *MovingDown* since it represents the moment when the arm starts movign down. For the rest, the system has the same components and the same dynamic equations.

- The model has a total of 22 variables. These variables have the following functionalities:
  - Real variables x, y, xd and yd represent the positions and velocities of the horizontal and the vertical arm.
  - Real variables x_{min}, x_{max}, y_{min}, y_{max}, m, k and b are constant variables that are used to represent the limits of the trajectories (the first 4 variables) and the dynamic parameters of the differential equations that the manipulators follow.
  - Boolean variables *up*, *down*, *inn*, *out*, *pick*, *drop*, ON, OFF and *act* represent some of the events. The first 4 represent the events *move_up*, *move_down*, *move_in* and *move_out*. The next 4 represent the two events of the picker and the two events of the sensor of the picker. The variable *act* is used to transition from state *Intermediate* to state *Movingdown* in the vertical manipulator.
  - Integer variable *count* has the same use that the one in the Stateflow model. It is used so that the vertical manipulator knows which signal has to send to the picker (either *pick* or *drop*) once the manipulators has reached the lower position.
  - Integer variable *res* is used to compute an internal operation with the variable *count* to decide which signal has to be sent to the picker.
- Variables are shared between state machines by using the *inner*, *outer output* and *outer input* notation that were explained previously.
- Operators *der*() and *previous*() are used to obtain the derivatives of a variable and its previous value (respect the actual time instant).
- As usual, every state machine has an initial state defined with the function *initialState* and several transitions defined with the function *transition*.

With this code that we just described, the Pick&Place station has been built and a simulation to verify its behaviour has been done. In this case the only variables that are used for the plotting visualization have been the variables x and y that represent the positions of the horizontal and vertical manipulators. The results of the simulation can be seen in Figure 6.3.

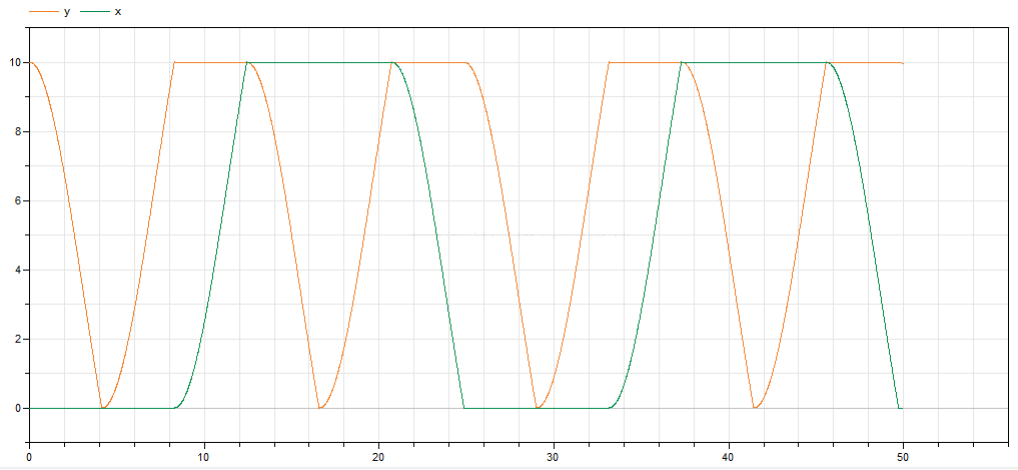

Figure 6.3: Results of the simulation of the model with Dymola.

As can be seen, the trajectories followed by the two manipulators during the time are very similar to the ones that we obtained with the other languages. Therefore, the controlled system behaves just as expected. With this validation ends the evaluation of the capabilities of the Modelica modeling language when it comes to model-based systems engineering with supervisory control development.

Chapter 7

SysML

The Systems Modeling Language (SysML) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information and procedures among others [24]. SysML is primarily used to combine 4 different aspects of a system including its requirements, behavior, structure and parametrics.

SysML has its roots from the Unified Modeling Language (UML). One of the objectives with SysML was to develop a modeling language that could be applied to a wide range of systems. This differs from its UML predecessor, which is more commonly used for software modeling. SysML represents a subset of UML with new features to specifically support systems engineering, including the addition of requirements modeling and enhancements to the structural representation of the system, among others. The interrelationship between SysML and UML can be seen in Figure 7.1.

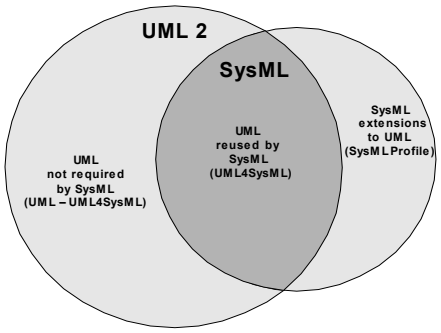

Figure 7.1: Overview of the SysML-UML relationship [24]

SysML includes a subset of the UML 2 and provides additional extensions to address new functional aspects for systems engineering modeling. SysML captures the multidisciplinary knowledge by providing various diagrams: block definition, internal block, parametric and package diagrams to present the structure of the system. It further delivers activity, sequence, state machine and use case diagrams to describe the behavior of the product. Finally, with its major contribution, it allows for modeling the requirements of a system with the help of its requirements diagrams. It also integrates the previous three aspects (i.e., structure, behavior, and requirements) through allocations across their corresponding elements. SysML further offers a profile mechanism, where a profile is formed from a set of stereotypes of its elements. These stereotypes extend the syntax of SysML allowing it to be more applicable in concrete applications.

SysML enlarges the scope of possibilities for systems engineers and adds several improvement with respect UML, which is more focused on the software side of high-tech systems. These improvements, as it can be seen in [55], include the following:

- SysML's semantics are more flexible and expressive. SysML reduces UML's software-centric restrictions and adds new diagrams/extensions that allow to model a wide range of systems, which may include hardware, software, information, processes, personnel, and facilities.

- SysML is a comparatively little language that is easier to learn and apply.

- SysML model management constructs support models, views, and viewpoints.

In this chapter we will review the potential of the SysML modeling language in terms of MBSE with supervisor synthesis capabilities. First, in Section 7.1 we will present the syntax and notation of the main elements that are used in SysML and its diagrams when modeling cyber-physical systems. Then, in Section 7.2 we will evaluate the capabilities of SysML in terms of model-based systems engineering with supervisory control synthesis development. Finally, in Section 7.3 we will try to build the same case study that we did with previous tools/languages.

7.1 Basic modeling concepts

As previously stated, SysML is a modeling language that provides a general purpose to support the specification, design, analysis and verification of complex system by using a subset of UML 2 with extensions. Its 4 main pillars include the modeling of the structure (Section 7.1.1), the requirements (Section 7.1.2), the behaviour (Section 7.1.3) and the parametrics (Section 7.1.4).

7.1.1 Structure

There are three types of diagrams for depicting the structural architecture including the block definition diagram, the internal block diagram, and the package diagram.

Block Definition diagram

The block definition diagram (BDD) in SysML is used to define the features of certain blocks in terms of their properties and operations, and the relationships between them such as the hierarchy of the system. The view is powerful for communicating domain, system and component structural hierarchy to a wide range of audiences. The basic notation of the block definition diagram can be seen in Figure 7.2.

Block definition diagrams mainly are made up of two basic components: blocks and relationships. These are the two only elements that can be found at the highest level of abstraction of the system. In addition to these two elements, a block definition diagram can also contain different kinds of ports, item flows and interfaces.

Blocks are used to describe the types of things that are part of the system, and the relationships are used to describe how these blocks are related. Each relationship can be used to relate one or two blocks so the possibility of relating a block to itself exists. A very used kind of block is the interface block, which is used to define interfaces for the blocks to interact. An instance specification defines an instance of a block.

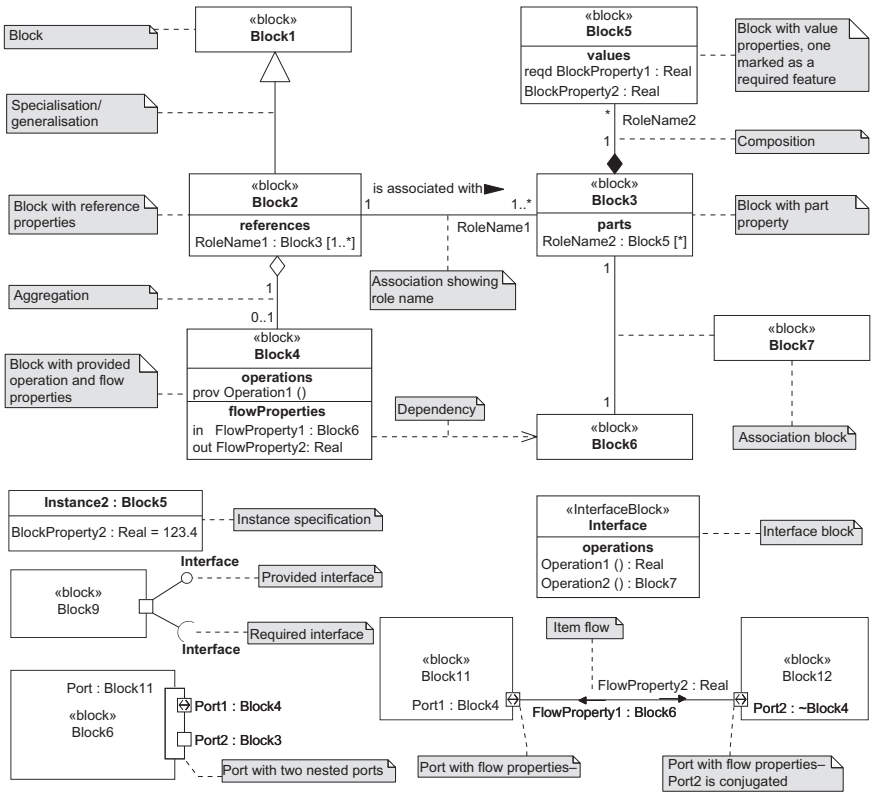

Figure 7.2: Notation of the block definition diagram [24].

Any block can have interaction points by defining ports. Each port is typed by a block and can be nested with other ports [24]. There are basically two types of ports:

- **full port**, which is used to represent an interaction point that has its own internal structure/parts and behaviour and, therefore, is a separate part of the model.
- **proxy port**, which is used to transfer the information about the features of its owning block to other external blocks.

To use along with the ports is the item flow, which is used to transfer a certain flow property (property of a block) between two ports. The different types of properties for a block are: part property, reference property, value property and flow property. Finally, to end with the breakdown of the components of the block definition, there are three main types of relationships:

- **Association**, which is used to represent a simple relationship between one or more blocks. It has two types which are referred to as aggregation and compositions which show shared parts and owned parts respectively.
- **Generalisation**, which is used to represent hierarchical relationships by showing the parent and child blocks.
- **Dependency**, which is used to represent the impact that a certain block may have on another block that depends on it.

Internal Block diagram

The internal block diagrams are used to model the internal structure of a block basically in terms of their properties and the connections between these properties. Some of these block's properties

allow to specify its values, parts and references to the blocks. One additional type of property is the port which is used to allow interactions between blocks. In the internal block diagram, an emphasis is put on the logical relationships between the different components rather than the structural breakdown itself. The basic notation of the internal block diagram is shown in Figure 7.3.

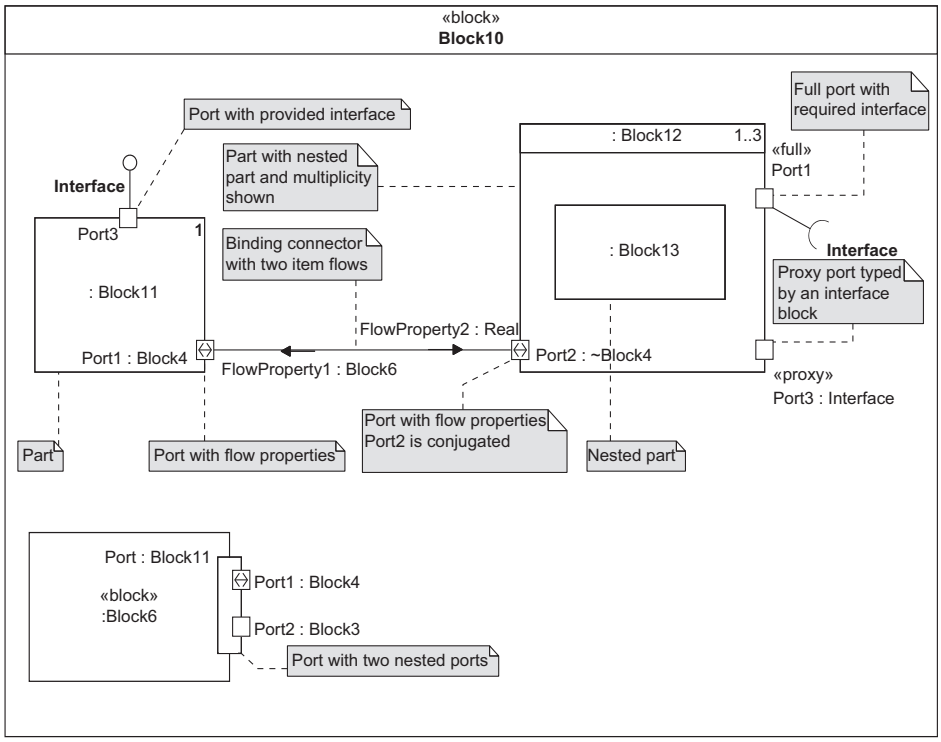

Figure 7.3: Notation of the internal block diagram [24].

The basic element that forms an internal block diagram is the part that describes blocks in the context of an owning block. An internal block diagram defines the different parts and their internal structures, showing how they are connected together through ports and showing the item flows that flow between them.

Just like with blocks, parts define interaction points through the use of ports which they come in the same two types: full port and proxy port. Also, a part can be connected to another part or itself by using a binding connector, where an item flow can flow across it. In addition to this, a part can be connected to a port on another part just like a port can be connected to zero or more ports.

Before moving to the next diagram, let us first consider the relationship between *internal block diagrams* and *block definition diagrams*, and therefore of parts and blocks. The first thing that has to be stated is that an internal block diagram is owned by a particular block. It is used when a block has an internal structure composed of other blocks and that allows the developer to focus on the connections between the blocks rather than the composition (block definition diagram) of each one of the blocks. For any block that is decomposed into sub-blocks an internal block diagram can be defined which is owned by that block.

The internal block diagram in 7.3 is owned by *Block10* and can be thought of as being its internal structure decomposition. *Block10* is shown as a containing block with the blocks that it is composed of shown as parts.

Package diagram

The package diagram is used to identify and relate together different sets of packages. A package contains a collection of diagram elements and implies some sort of ownership. They can be used on the package diagram as well as on other diagrams and they can be related to each other through the use of different types of dependency relationships. The basic notation of the package diagram is shown in Figure 7.4.

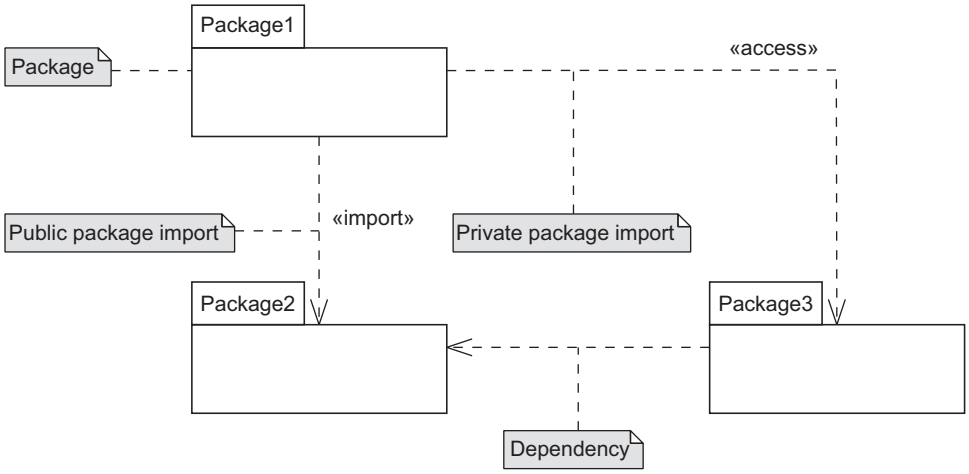

Figure 7.4: Notation of the package diagram [24].

As one can see, the two main elements of a package diagram are the package and the dependency. Packages are used to structure a model in a similar way like directories organize files in a computer. On the other hand, there is one type of dependency and that is the package import. This can be further divided into the public package import and the private package import. The type of import package (public or private) infers on the visibility of the information that is being imported from the package.

A package import means (regardless of the type) that the package which is being pointed to (referred to as target) is imported into the other package (referred to a source) but with the target package remaining its own package.

7.1.2 Requirements

One of the major improvements SysML with respect to UML 2 is its support for representing requirements and relating them to the model of a system. SysML has a dedicated *requirement diagram* that is used to represent requirements and their relationships. The basic notation of the *requirement diagram* is shown in Figure 7.5.

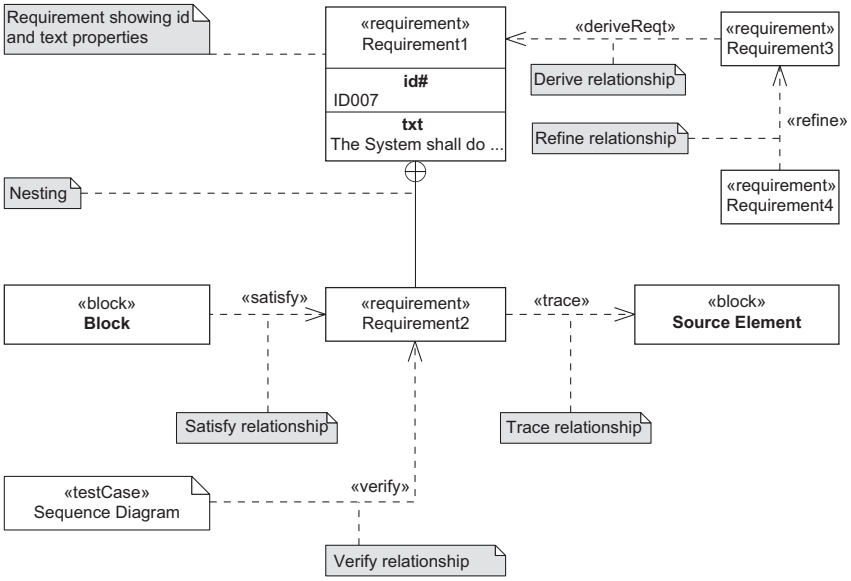

Figure 7.5: Notation of the requirement diagram [24].

The requirement diagram has three basic components which are the requirements, its relationships and the test cases. A requirement is used to specify a capability or functionality that the system must (or should) satisfy. That can be defined as a function that the system must perform or a performance condition that the system must achieve. The requirement diagram that is being described in this section can depict the requirements in a graphical, tabular or tree structure format. The relationships are used to relate the requirements to each other or to other elements (like blocks) and the test cases can be linked to the requirements to show how the requirements are verified.

The different types of relationships that SysML provides for the requirement diagram are:

- **Satisfy relationship**, which is used to relate elements of a design or implementation model to the requirements that they are supposed to satisfy.
- **Trace relationship**, which is used to allow model elements and requirements to be related to each other (to imply a condition on a certain parameter of the model elements).
- **Refine relationship**, which is used to show how model elements and requirements can be used to further refine other model elements and requirements.
- **Verify relationship**, which is used to ensure that a certain test case verifies a given requirement.
- **Derive relationship**, which is used to relate a derived requirement to its source requirement. The derived requirements correspond to the requirements at the next level of the hierarchy of the system.

These various types of relationship allow the modeller to explicitly relate different parts of a model to the requirements as a way of ensuring the consistency of the model.

7.1.3 Behaviour

This subsection is used to specify the dynamic, behavioral constructs that are used in SysML's behavioral diagrams of the system or an element in its own domain. This is done by using the

following 4 diagrams: the activity diagram, the sequence diagram, the state machine diagram and the use case diagram.

Activity Diagrams

The activity diagrams are the ones that model the lowest-level part of the behaviour of the system in comparison with the rest of behavioural diagrams. Just like state machine diagrams model the behaviour within elements, activity diagrams are used to model the behaviour within a certain operation or process. The basic notation for the activity diagram can be seen in Figure 7.6.

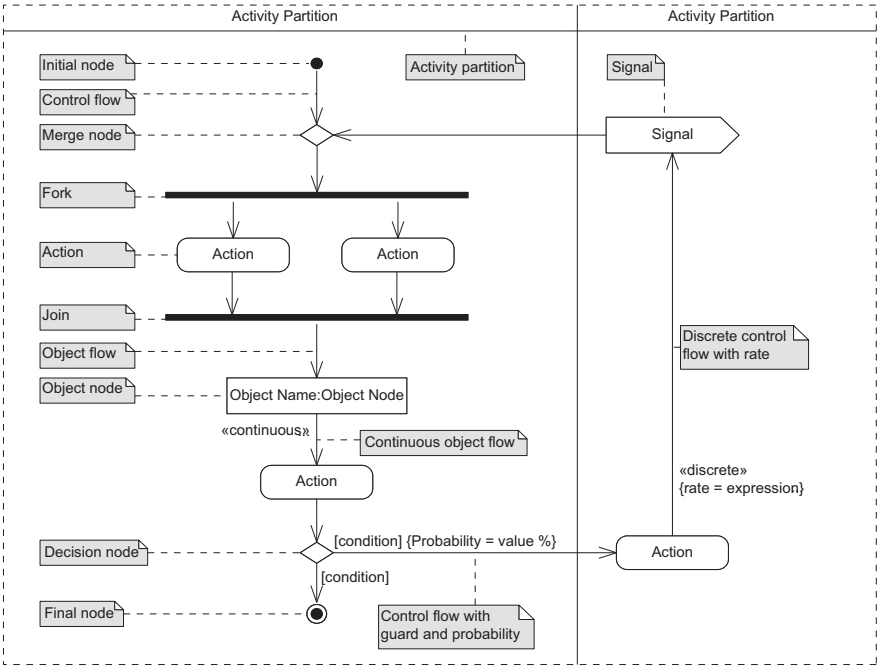

Figure 7.6: Notation of the activity diagram [24].

The activity diagram has three main elements which are the activity nodes, activity edges and regions. Activity nodes can be divided in three: action, object and control node. The action element is the focus on these diagrams since it represents a possible behaviour. There are several types of actions available but they will not be introduced in the scope of this thesis. Any action has the possibility of having a pin that can be used to show the object flow that an object carries. When it comes to control nodes, they can be grouped in the following:

- The initial/final nodes are used to represent when the activity starts/ends. In addition to this, the flow final node is used to terminate a particular flow without finishing the diagram.
- Flow can be split into several paths and then re-joined together at a particular point by using the fork and join nodes.
- A decision node which permits a flow to branch into different routes according to several different guard conditions and a merge node which allows to join different flows back together into a single one.

Activity edges are used to connect an activity node to another one or also to itself. It has two main types: the control flow and the object flow. The first one is used to represent the main routes that exist in the activity diagram and that connect one or two activity nodes. On the other hand, the second one represents the information that flows between one or more activity nodes

and does it by carrying the object type of the activity node. When it comes to representing flow of information, SysML uses these three elements: the object node, the event and the signal. The first one is used to model the information that has been represented elsewhere in the model by a block and which is forming an input to or an output from an action. The event and the signal work complementary. The first one represents an input event in the activity diagram and the second one an output event leaving the diagram. They correspond to receipt events and send events of a state machine diagram.

Finally, the other main element in an activity diagram is the region. It is divided into two subtypes: the interruptible region and the activity partition. The first one represents a boundary that can be introduced in the diagram that contains all the actions that can be interrupted. The second one is used to allow different actions to be grouped together for a certain reason, usually to show responsibility among the actions.

Sequence diagrams

Sequence diagrams are used to model message-based exchange behaviour of the system. That is basically the definition of the flow control between the actors and the blocks of the system or the interaction between different parts of the system. It provides with mechanisms that support highly complex interactions with special constructs to model several types of control logic, reference interactions on other sequence diagrams among others. The basic notation for the sequence diagram can be seen in Figure 7.7.

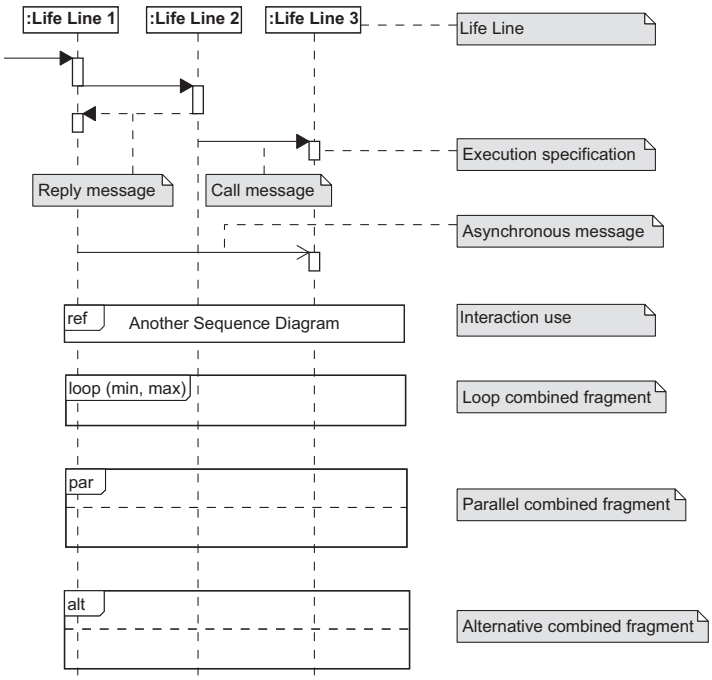

Figure 7.7: Notation of the sequence diagram [24].

There are two main elements in sequence diagrams which are the life lines and the messages. Additional elements can be used to permit other diagrams to be referenced, interaction uses and constructions such as looping and parallel behaviour to be represented using combined fragments.

Life lines are used to model any participant that takes place in a scenario over the execution time. It will refer to an element from another aspect of the model like a block or an actor. The interaction sequence between life lines is done with the sending and receiving messages which are drawn between life lines.

Complex Scenarios can be represented containing looping, parallel and alternative behaviour, shown using various types of combined fragment. In addition to this, a sequence diagram can refer to another via the interaction use notation, allowing more and more complicated scenarios to be developed. However, it is worth noting that the various combined fragment notations can be nested, allowing very complicated scenarios to be modelled. In particular, the use of the alternative combined fragment notation allows alternative paths through a scenario to be shown.

State Machine Diagrams

State machine diagrams are used to model the event-based behaviour of the system. They model the changes of the system and the logical conditions under which they happen for instances of blocks, which are known in SysML as instance specifications. They realise such behaviour by relating it to states that the components of the system, modelled as blocks, can be at any given time. They concentrate on the events that cause that change of the state (known as transition) of the system and the behaviour is associated with that transition or inside the new state. The basic notation of a state machine diagram can be seen in Figure 7.8.

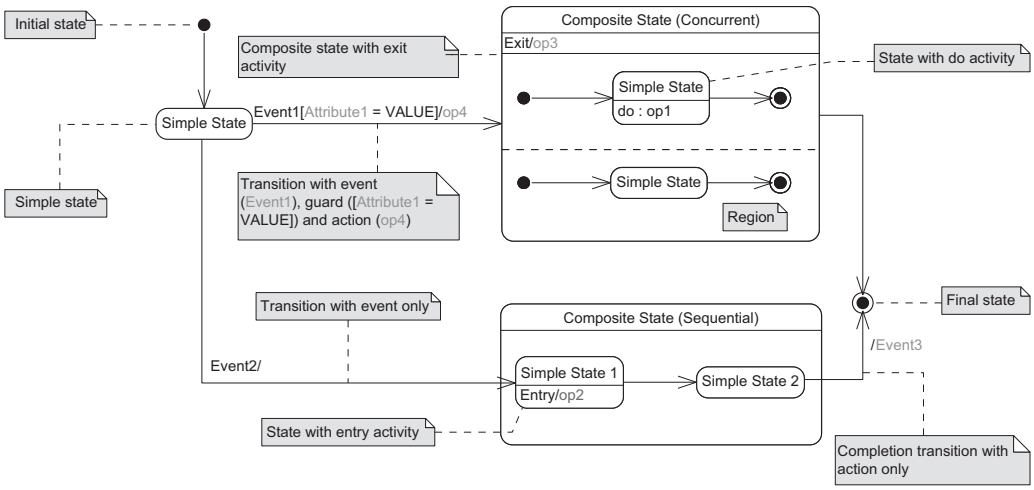

Figure 7.8: Notation of the state machine diagram [24].

Overall, state machines diagram have to basic components: states the transitions between them. States show what is happening at a particular instant in time when an instance specification typed by a block is active. Transitions are used to change between the different states that a system has.

In SysML, there are four types of state: initial state, simple state, composite state and final state. All these types of states have the same nature than in MechatronicUML. Each state allows for an activity to be defined even though it is not mandatory for a state to have one. An activity represents a non-atomic unit of behaviour which is related to the operations on a block and that can be interrupted during its execution. SysML supports entry activities, during activities and exit activities as the types of activities to be performed in a state. As stated, composite states have the same nature than in MechatronicUML: they are states that have their own state machine diagrams inside.

The syntax for transitions is also very similar than with Stateflow or MechatronicUML. Each transition can have a boolean condition (also referred to as guard condition) that must evaluate to true for the transition to fire. Usually this guard condition relates to certain properties of a block. A transition can also have an action, which is an activity whose behaviour is atomic and therefore can never be interrupted and will always be completed. Finally, a transition can have a trigger event that may also be needed for the transition to fire. This type of event is also known as receipt event. Per each receipt event, there has to be a complementary send event. A send event

represents the broadcast of a message being send from a state machine to another. In SysML, it is generally assumed that a send event is broadcast to all elements in the system and therefore each of the other components has the possibility to receive and evolve according to that event. A send event is usually modelled as the action on a certain transition.

Use Case diagram

The use case diagram is used to realise the behaviour of the system but with an emphasis on the functionality rather than the control and logical timing of the system during its execution. This diagram represents the highest level of abstraction in terms of behaviour that is available in SysML. The notation that is used on the use case diagrams is shown in Figure 7.9.

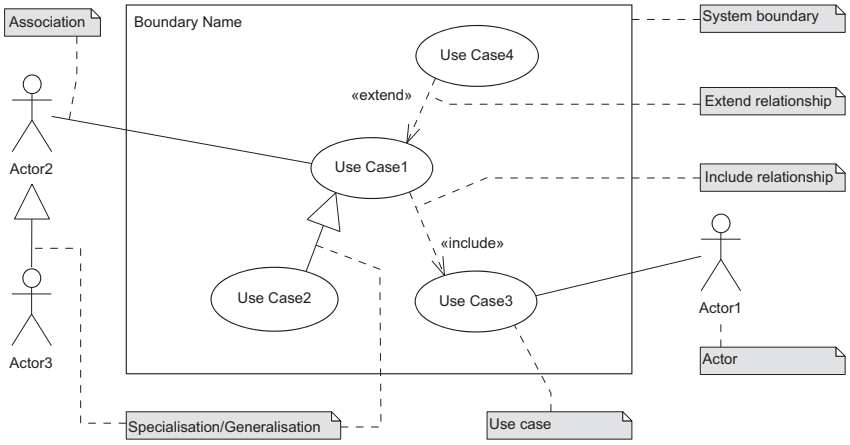

Figure 7.9: Notation of the use case diagram [24].

There are four basic components in the use case diagrams and these are the use cases, the actors, the relationships and the system boundary. The only mandatory component for a use case diagram is the use case while all the others are optional. Each use case represents the behaviour of the system that has an impact on an actor. An actor represents the role of a person, place, thing that interacts with the system and that is impacted or has an interest on it.

Use cases are related to actors and to other use cases using a number of different types of relationships:

- **Association relationship**, which is used to relate use cases to actors.
- **Include relationship**, which is used to represent when a certain functionality may be split from the main use case to be used by another use case.
- **Extend relationship**, which is used to extend the functionality of the use case in some way. This usually is due to the fact that the functionality of a use case may change during the execution time of the system.
- **Specialisation/generalisation relationship**, which are used in the same as in a block definition diagram. Specialization is used when one use case is a specialization of another and generalisation between use cases allows inheritance of behaviour and relationships.

Finally, the last element that can appear in a use case diagram is the system boundary. This is used to describe the context of the system. Everything that is inside the system boundary is part of the system and everything outside is external to it. Actors are components that are always outside the system boundary and, therefore, each interaction between an actor and a use case indicates that there is an interface between the actor and the system. System boundaries are not mandatory on a use case diagram.

7.1.4 Parametrics

Parametrics is the aspect of modeling that deals with defining constraints to give system parameters or value properties additional meaning in the model. In SysML this is done with the constraint block and associated parametric diagram which allows the definition and use of networks of constraints which represent the rules that constrain the properties of a given system. The most common examples of parametrics can include: a certain law that a group of variables/parameters must satisfy (e.g. Newton's laws) or inequalities that a group of variables/parameters must fulfill (e.g. the maximum speed of the car has to be 120 km/h).

The parametric diagrams are basically made up of three components that are the constraint blocks, the parts and the connectors. The only component that is always needed is the constraint block while the other two are optional. Constraint blocks can be connected to constraint blocks and to parts through the use of connectors and, even though they are used in the parametric diagram, they are defined in a block definition diagram.

There are two important aspects regarding parametric constraints in SysML: their definition and their use. The notation for the definition of the constraint block is shown in Figure 7.10.

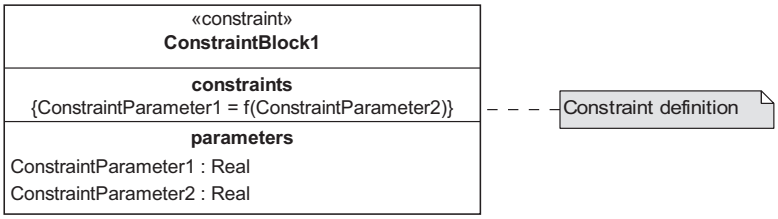

Figure 7.10: Notation of the parametric diagram - definition of the constraint block [24].

A constraint block is defined using a block which has two main compartments that are the constraints and the parameters compartment. The constraints compartment contains an equation/-expression/law/rule that relates together all the parameters that are given in the parameters compartment.

Such constrain blocks are defined on a block definition diagram. Even though constraint blocks are defined on a block definition diagram, such definitions are not mixed with regular blocks on the same diagram. Once defined, constraint blocks can be used any number of times on one or more parametric diagrams diagrams, the notation for which is shown in Figure 7.11.

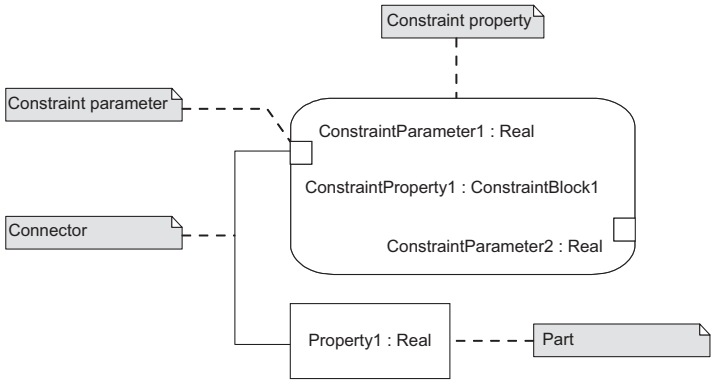

Figure 7.11: Notation of the parametric diagram use of constraint block [24].

Each constraint block can be used multiple times on a parametric diagram. The use of a constraint

block is shown as a round-cornered rectangle known as a constraint property. Each constraint property is has to be named:

Name : Constraint Name

This allows each use of a constraint block to be distinguished from other uses of the same constraint block. Small rectangles attached to the inside edge of the constraint property represent each constraint parameter. These are named and their names correspond to the parameters defined for the constraint block in its definition.

These constraint parameters provide connection points that can be connected, via connectors, to other constraint parameters on the same or other constraint properties or to block properties. When connecting a constraint parameter to a block property, this block property is represented on the diagram by a rectangle known as a part.

7.2 MBSE capabilities

7.2.1 Specification phase

Before starting with the evaluation of the capabilities of the SysML language in the specification phase, it should be interesting to explain how the basic modeling constructs of simulation tools like Simulink (models, components, ports, links and dynamic equations) translate to SysML constructs. To start with, models in simulation tools are blocks in SysML which have an internal structure and that are not a component of other blocks. These blocks are defined with the block definition diagram. The internal structure of these blocks (which relates to the block diagrams of models in Simulink) are modeled with internal block diagrams. In the same way, a subsystem in Simulink corresponds to a SysML block with internal structure and an atomic block in Simulink corresponds to a SysML block without internal structure.

In addition to this, block properties such as parts, connectors, values, ports (and flow properties to be more specific) or constraint properties in SysML correspond to the components of a model (in Simulink) and their interactions. Being more specific:

- Parts correspond to blocks in a Simulink model
- Connectors correspond to the links that are used to join different blocks. However, connectors in SysML miss some functionalities like the direction of the signal-flow (which is not defined).
- Value properties correspond to the variables and parameters that are used in the blocks of the block diagram. However, the value properties of SysML cannot distinguish between variables (variable value) and parameters (fixed value), and between continuous and discrete.
- Flow properties are used as the interaction points (input/output ports) of blocks in the block diagram. However, there are some semantic inconsistencies since flow properties specify things like the kind of things that flow (which the interaction points do not) and the interaction ports specify things like the flow rate (which the flow properties do not).
- Differential/difference equations can be described by using constraint blocks with constraint properties that represent these mathematical constraints (operator *der()* used to model the derivative of a variable).

Having said that, it is clear that the low-level time-driven part of the system can be modeled by using the block definition diagram, the internal block diagrams and constraint blocks from the parametric diagram.

On the other hand, when modeling the high-level event-driven part of the system there are some things that need to be considered. First of all, and the most important, the SysML language

does support the automaton concept (with basic features like states and transitions) but it does not support the hybrid automaton formalism since the states in the state machine diagrams do not provide ways to represent the continuous dynamics in a particular state. The dynamics for a certain set of variables are defined in a constraint block and therefore are unique and cannot be changed. As a result, switching between operational modes is not possible since it is not even possible to describe all the operational modes in terms of their dynamics (even though there might exist certain states that represent those operational modes).

On the other hand, events are easily modelled in the state machine diagrams by either declaring them as receipt or send events depending on their source/destination. As it has been seen in the previous section, event broadcasting is possible since at any time a send event is defined as the action of a transition and this transition broadcasts this event to any state machine in the system so that it has the possibility to evolve. Because of this reality, synchronization of events cannot be guaranteed since every time an event is broadcasted it will only make the other state machines evolve if they are at a particular state that is fired for that particular event. The concept of final states is supported and can be declared for in any state machine diagram.

Just like with the continuous-time equations, discrete variables cannot be declared in any particular state and, therefore, shared variables, initialization of variables and updates of variables in time are not supported. Finally, the synchronization in time exists between all the constraint blocks but this approach differs from the other languages approaches where the synchronization in time is between the different dynamics of the different states that are active during the execution of the system.

That is all the evaluation in terms of modeling the uncontrolled plant. When it comes to modeling requirements for the system, SysML offers some features. However the type of requirements that can be modelled with SysML are different than the ones that are modelled with the CIF3 toolset. With CIF3, the developer creates models in the form of state machines for the high-level requirements of the system. On the other hand, in SysML requirements are always related with the low-level part of the system, so they are related with the variables that are used in the constraint blocks. Examples of requirements in SysML would be the maximum time that a car has to take to accelerate from between two velocities or the maximum flow rate that can go through an hydraulic pump. Therefore, since the nature of both requirements is not the same, it is better to say that SysML does not offer any features for modeling the requirements of the high-level system to avoid phenomena like a deadlock.

7.2.2 Supervisor development phase

From what we have seen in the last section, after realizing the nature of the requirements that are specified in the requirement diagrams, it is obvious that SysML cannot provide with any synthesis algorithm for the development of any sort of supervisory controller. However, just like with the other languages/tools, there are some possibilities for manual development of the supervisory but there are some things that have to be pointed out.

Just like in Stateflow or Modelica, by modifying the transitions between the states in the state machine diagrams, the developer can obtain a controlled system with (sometimes) the desired behaviour. However, there are some things that are done differently in SysML and have to be explained. First of all, we have seen that event broadcasting is possible thanks to the receipt/send events that can be used in the diagrams as part of the transitions of the states. But, it is important to note that events can only be send as an action of a transition and not like an activity in a state. Remember that activities in states (regardless of their type) are always associated with operations on a block. Therefore, some possibilities that tools like Stateflow offer are not possible with SysML. In addition to this, since mode switching is not possible in terms of changing the dynamic equations, some of the guard conditions might never be enabled and therefore this could make the

design fail already. Limitations like this might have to be considered whenever a developer wants to study a certain cyber-physical system with SysML.

7.2.3 Validation phase

SysML is a modeling language with several strong modeling capabilities, but just as it happens with Modelica, it does not provide a simulation platform to perform dynamic solver-based simulations. These missing capabilities are crucial for the validation phase of the systems engineering process and therefore support for these capabilities needs to be provided.

Currently, nowadays, simulations and optimizations are conducted in separate and particular simulation tools. The two main combinations between SysML and simulation tools during the last years have been with the Simulink toolset and the Modelica language. On the one hand, In [30], the authors propose an extension of SysML which enables description of continuous-time behavior. They also develop its execution tool integrated on an Eclipse-based platform by exploiting co-simulation of SysML and Matlab/Simulink. Another integration between SysML and Simulink was done in [48], where the authors use SysML to capture the architecture of a system, and to transform the behavior representation of the architecture into a Simulink model ready to simulate.

On the other hand, a SysML-Modelica transformation is described in [6], where a SysML profile is defined to represent the Modelica constructs. This transformation was standardized by the Object Management Group and the specification can be found in [37]. We would like to point out that some previous integration efforts were made with respect to the SysML version 1.2.

Just like with simulation-based validation of the models, SysML by itself does not provide any possibilities in terms of formal verification of the models developed with the language. Way less effort has been put in this, in comparison with the SysML transformations to simulation tools, but there is some work to be mentioned. In [34] an approach was introduced to support the modeling and verification process of SysML models with the objective of verifying the requirements before the implementation. In [41] a SysML-based environment named AVATAR is presented and which has a direct translation to the well established UPPAAL toolset for model checking and formal verification.

7.2.4 Implementation phase

Once again, since SysML is a modeling language and not a toolset, and therefore there are no possibilities available for the Systems Modeling Language in terms of code generation for platform implementation. However, just like in the previous phase, there are some possibilities of generating code from SysML models by using other modeling tools that support the SysML language. One possibility could be to use the IBM Rational Rhapsody Designer toolset [27] for Systems Engineers which supports fully the SysML language. It provides code generation for C, C++ and Java. Another viable possibility could be the Enterprise Architect Systems Engineering Edition toolset [45] which is another tool that supports the SysML modeling language. It provides code generation for C, C++, C#, Java and VBNet. This last one (Enterprise) might have SysML version limitations (code generation for SysML up to a certain version).

7.3 Case study

As we have seen in previous sections when building the case study with different languages/tools, there are components that have states with dynamic equations in them that refer to the operational mode in that particular state. These are the cases of the two arm manipulators when they are not standing still. However, as we have seen when does its evaluation, SysML does not support the concept of hybrid automata because the dynamics cannot be associated with states. Instead, they

have to be defined in the constraint blocks of the parametric diagram where they remain static following a single law that cannot be changed. This is definitely a problem since the trajectories of the two manipulators change their directions and, therefore, their initial conditions and there is no way to switch between conditions during the execution of the system. One could thing that this could be solved by using two sets of variables to model the two directions of the movement (moving up and down or out and in) but this would not solve the problem since the dynamics run all the time and therefore the resulting system would not be the desired one. It would be a system that is moving regardless of its state and it would also not respect the physical limitations of the manipulators (minimum and maximum values for the positions).

In addition to this, it has been stated that the state machines from SysML lack in a lot of the important concepts for developing supervisory control structures that have been mentioned in this thesis. Not being able to provide discrete variables or the impossibility of having event synchronization really hurts the chances of developing a model of the xCPS platform with SysML.

To conclude with, with the features that the modeling language SysML provides nowadays, it is not possible to develop a (not so complex) system like the xCPS platform with its correct dynamics and the supervisor that ensures the desired behaviour.

Chapter 8

Comparison between modeling languages/tools

The previous four chapters have been used to introduce, describe, analyze and evaluate (within a certain criteria) the modeling languages/tools Simulink&Stateflow, MechatronicUML, Modelica and SysML. This process has been done individually for each one of the languages but it is also very important to see where they stand when compared against each other and see who outperforms who in which subjects. This chapter is fully dedicated to compare the four languages/tools extensively so that any possible future user can realize of the scope of each one and their capabilities. To do this, we have divided it in two main sections. In 8.1, the overall comparison between languages/tools will be done with a table that contains all the tested concepts that have been used. Apart from the table, several qualitative explanations will be added to complement the information that the table does not provide and to allow the reader to have a better idea of the true potential of each tool/language. In 8.2, a final review of the different tools/languages will be done.

8.1 Overall comparison

Each one of the modeling tools that has been evaluated in this thesis was created for a similar purpose but the methods that they use and the scope of their real capabilities is very different. Therefore, it is just as important to know if a language provides/supports a certain concept than the scope of the capabilities of the same language regarding the same concept.

To start with the overall comparison, Table 8.1 has been built to collect all the information regarding the concepts that each modeling language/tool features or not. This is an easy way to see if any concept is not supported by a certain language and to see the first differences in terms of capabilities between the different tools. After introducing the table, a discussion will be done for every single one of the development phases and between all the different languages/tools. This table can be seen below:

	Modeling languages			
	Simulink&Stateflow	MechatronicUML	Modelica	SysML
Specification phase				
Time-driven system				
Continuous-time	✓	✗	✓	✓
Discrete-time	✓	✗	✓	✓
Event-driven system				
Automata	✓	✓	✓	✓
Mode-switching	✓	✓	✓	✗
Events	✓	✓	✗	✓
Final states	✗	✓	✗	✓
Discrete variables	✓	✓	✓	✗
Updates of variables	✓	✓[1]	✓	✗
Synchronization in time	✗	✗	✓	✗
Synchronization of events	✓	✓	✗	✗
Shared variables	✓	✓	✓	✗
Initialization	✓	✓	✓	✗
Requirements	✗	✗	✗	✗
Supervisor development phase				
Manual development	✓	✓	✓	✓
Synthesis algorithm	✗	✗	✗	✗
Validation phase				
Simulation-based visualization	✓	✓[2]	✓[3]	✓[4]
Formal verification	✗	✓[5]	✗	✓[6]
Implementation phase				
Code generation	✗	✓	✓[7]	✓[8]

[1] Only for discrete variables.
[2] A transformation to an external tool is required.
[3] With OpenModelica or another simulation environment.
[4] A transformation to an external tool is required.
[5] A transformation to an external tool is required.
[6] A transformation to an external tool is required.
[7] With OpenModelica or another simulation environment.
[8] A transformation to an external tool is required.

Table 8.1: Comparison between the different modeling languages/tools.

Specification phase

The first differences between the different languages tools appear in the specification phase and, in particular, when modeling the time-driven part of the system. As we have previously seen, the atomic components that MechatronicUML uses to model components with time-driven behaviour do not have an own behaviour specified since this has to be specified by a outside tool like Simulink or Modelica. Therefore, MechatronicUMl's capabilities in this field are only the communications between the interfaces of these components and the rest of components in the system. On the other side, Simulink's and Modelica's capabilities when modeling continuous/discrete-time systems are very powerful because of the large amounts of libraries that each one has dedicated to this subject. The only difference between them is the modeling approach. Simulink uses a block diagram approach and Modelica a graphical object-oriented equation-based approach. Both approaches are very intuitive when it comes to modeling but Modelica's is more intuitive because, since it's object-oriented nature, the developer can see the physical appearance of the system that is being modeled. Finally, SysML uses the block definition diagram, the internal block diagram and the parametric diagram to model this part of the system. The models are divided in blocks (defined

with block definition diagrams) which may have an internal structure (defined with the internal block diagram) and whose dynamic behaviour can be specified using the constraint block of the parametric diagram when applied to certain properties of the blocks.

Switching to the event-driven part of the system, the differences between languages/tools and its magnitudes increase significantly. The automata formalism is supported by all the languages/tools but the scope of its functionality really depends on the case. A basic concept like the events (result of the abstraction process) is not supported in Modelica. We have to remember that events in Modelica correspond to the physical reality of the events at the low-level part of the system. On the other hand, Stateflow and MechatronicUML support events in a different ways: Stateflow can declare them as input, outputs or locals to a chart and MechatronicUML uses messages to be exchanged between the statecharts of the discrete components. Eventually, SysML supports the automata concept (even though not the hybrid automata) and also the concept of events thanks to the receipt/send event entities that are used in the state machine diagrams. Among all the tools/languages, Stateflow and Modelica do not offer the possibility of defining final states in its charts and even though this is not a huge inconvenient, it needs to be remembered.

When it comes to variables, all the modeling languages/tools (except the SysML language) support discrete variables and their updating method upon time. As previously stated, there is no possibilities in SysML when it comes to defining continuous-time behaviour or discrete variables (and their corresponding updates in time) in the states of the state machine diagrams. In addition to this, any concept regarding discrete variables nor mode switching is obviously not supported in SysML. Going back to the other 3 languages/tools, this case changes when the variables have a continuous-time evolution. In Stateflow, variables can have an evolution in three cases: if they come from a Simulink block that has continuous-time dynamics, if they come from a Simulink function that runs in continuous-time, when is created in a Stateflow chart with an *update method* as *continuous-time*. Modelica's state machines can have continuous-time evolution thanks to the extension that we presented earlier. On the other side, MechatronicUML does not have variables that evolve in continuous time since this variables come from components with time-driven behaviour and these are modeled in an external tool. Variables can be shared among statecharts of the different languages/tools but in different ways. In Stateflow and in Modelica this is done through the hierarchical structure of the charts and in MechatronicUML it is done through the ports that communicate the components. Finally, the initialization of the variables is mostly very similar in all the languages/tools. In both Simulink&Stateflow and Modelica, the variables need to be initialized before the chart is instantiated and the same can be done in MechatronicUML even though is not mandatory.

In another subject where the languages kind of differ is in terms of synchronization in time. First of all, synchronization in time is not supported in MechatronicUML because of the nature of its components (they have discrete event-driven behaviour). This concept is also not supported in Stateflow since the different parallel/exclusive (OR) components run according to a certain order of execution. On the other side, the state machines in Modelica do synchronize in time since they usually have the same clock. In SysML, since there is no possibilities of declaring continuous-time behaviour in states, there is obviously no support for synchronization in time of continuous variables.

The last thing in the specification phase in which all the languages (except SysML) share similarities is that none of them offer any possibilities to model the system's requirements either in terms of automata or in another way. We have to remember that the requirements that are modeled with the requirement diagram of SysML always represent goals/functions which are always related with a set of constraint blocks of the system.

Supervisor development phase

In the supervisor development phase is where the tested languages share more similarities. That's because none of the languages provides with any type of algorithm to apply the synthesis technique

to obtain a supervisor. Therefore, each one of the languages/tools has to offer a manual procedure to develop the desired supervisor to control the system. Here is where the similarities end.

On the one side, the method that Stateflow uses is based on the translation of the requirements into changes in the main chart. This includes the execution of actions (that are mostly used for updating variables) or the broadcast of events in any point in the chart to allow the evolution of a certain state machine. This actions can be performed in the transitions or in the states (*entry*, *during* and *exit* actions). This approach is quite similar to the one that can be done with Modelica. However, we have to remember that the scope of functionalities that are provided with Modelica in terms of the supervisory controller are less than in Stateflow. This is basically due to the restriction that Modelica has in terms of actions and transitions. On the other hand, MechatronicUML builds the supervisory controller with asynchronous message communication between the discrete ports of the components by following a certain Real-Time Coordination Protocol. This method, in comparison with the one used with Stateflow or Modelica, is more structured as a method since the steps are always the same (specify the roles, their behaviour and properties, the role connector and the messages that are going to be exchanged). However, in terms of the functionalities and supervisory control possibilities, the MechatronicUML method offers more or less the same as Stateflow. Eventually, SysML uses a notation for event-exchange similar to the one used in MechatronicUML by using the receipt/send events to trigger transitions and to allow implement the desired control logic.

Validation phase

The differences between languages/tools in the validation phase are strictly related to the simulation environment that each one uses. To start with, it has to be noted that MechatronicUML does not have a simulation tool by itself. However, as it has been seen, there exists several transformations that allow the developer to validate his models/controllers. One way is through model-checking via the model transformation to the model checker UPPAAL and the other way is through simulation-based validation with transformations to Simulink (with the Simulink Adapter) or to Modelica (with the Real-Time Coordination library).

In contrast with MechatronicUML, Simulink&Stateflow and Modelica (through the use OpenModelica) have their own graphical simulation environment. Both of the are based on a graphical user interface that provides with multiple features and possibilities either for the modeling process or for the simulation afterwards. Both of them are very user-friendly and intuitive to use and provide with the necessary tools to perform a complete analysis of the system behaviour by allowing to change the set of simulation parameters and plot the results in different ways. Therefore, each one these two tools/languages provide a simulation-based validation platform. On the other hand, the two tools/languages also share a lot of similarities when it comes to the formal verification of the models. With Simulink&Stateflow, despite the efforts that have been put in the area, proper formal verification methods that support all the functionalities are still to be developed. In the case of Modelica, as previously stated, because of some of the modeling practices that the language has, there is no possibility to support formal verification methods.

Eventually, just like it happens with Modelica, since SysML is a modeling language and not a tool, SysML needs of a simulation tool or a tool for formal verification case. In the first case, the two main possibilities are two transformations for SysML models to be executed and simulated in Simulink and Modelica (and its respective simulation environment). In the second case, when it comes to formal verification, there are fewer possibilities but the most outstanding is the SysML-based modeling environment AVATAR that has a direct translation of its models to the model-checker UPPAAL for straight formal verification.

Implementation phase

In the implementation phase, just like it happens with the validation phase, the main differences come from the nature of the tools/languages. Tools like Simulink&Stateflow or MechatronicUML are available to generate code for a certain specific platform by themselves (with extensions of the

tools like the Simulink Coder) and languages like Modelica or SysML need of other toolsets for the implementation phase. Starting with the tools, Simulink&Stateflow are able to generate code with help of the extension Simulink Coder or Embedded Coder for C, C++ while MechatronicUML can generate code for Real-Time Java and C++. On the other hand, Modelica can generate code by using the OpenModelica environment for C, CSharp, C++, Adevs, sfmi, XML, Java and Javascript. Finally, for SysML there are some options but the most remarkable are the IBM Rhapsody Designer toolset which generates code for C, C++ and Java and the Enterprise Architect Systems Engineering Edition toolset which generates code for C, C++, C#, Java and VBNet.

8.2 Comparison review

8.2.1 Simulink&Stateflow review

The MATLAB toolbox with Simulink and Stateflow provides excellent modeling and simulation capabilities for complex cyber-physical systems that mix time-based and event-based domains. From the start of the design in the specification phase till the code generation in the implementation phase, this toolset provides with a very wide variety of features that support the whole systems engineering process. All the modeling process is supported by a very friendly and easy to use graphical user interface that helps the developer create almost any system with the desired functionality that it requires.

From the 4 modeling languages/tools that have been reviewed in this thesis, Simulink&Stateflow bring the most complete overall package for the MBSE process. Simulink is mainly used to model the low-level part of the system (its inner structure and behaviour) through the definition of block diagrams. For that purpose, Simulink provides with a large number of libraries which contain several blocks that allow the developer to model almost any physical reality that a system can experience. This approach, when used correctly, results in models that can combine several engineering disciplines such as mechanical, electrical, electronic or control in a block-based fashion. On the other hand, Stateflow provides with a wide variety of features to model and implement the supervisory control logic that is used at the high-level of the system. This can be achieved in different ways but the most common is by using the Stateflow's charts to represent the state machines that end resulting (after all the modeling process) in the supervisory controller and the controlled hybrid plant.

As previously stated, Stateflow models are a type of Simulink blocs and, therefore, the process of integrating both models (Simulink's block diagram with Stateflow charts) is done almost instantly. That is all for the specification phase and supervisory controller phase.

For the validation phase, the Simulink&Stateflow toolset by itself only provides one of the two main approaches (simulation-based visualization and formal verification) for validation of the behaviour of the controlled system. The Simulink toolbox provides with one of the most powerful simulation environments that is out there for modeling tools which allows for different analysis techniques and simulation possibilities. One the other hand, the only main drawback for the Simulink&Stateflow toolset is the lacking of a formal verification method for its models. The lack of formal semantics of the models used in this toolset has created a strong need to develop automatic semantic translators that can translate Simulink&Stateflow models into models of different analysis tools for the formal verification or model-checking process.

Finally, looking in to the implementation phase, it is true that the Simulink&Stateflow toolset do not provide any possibilities of code generation by themselves. However, the two toolboxes used for this purpose (Simulink Coder and Embedded coder toolbox) are usually included in the list of toolboxes provide with MATLAB when a company purchases the software. Therefore, a lot of the times the code generation and implementation process is supported without having to buy the new toolboxes in addition to the software.

8.2.2 MechatronicUML review

MechatronicUML is a modeling tool that is mainly used to model and develop the software that is embedded in the high-tech cyber-physical systems that are found in today's world [47]. As we have seen, this approach does not provide any possibilities when it comes to modeling and simulating the time-driven low-level part of the system including the physical components and the control strategies. Therefore, this reality should be the first thing to consider when choosing MechatronicUML as a tool for model-based systems engineering.

However, one the other hand, this tool offers several capabilities when it comes to the high-level of the system. Through the use of Real-Time Statecharts to model the discrete-event behaviour of atomic components and Real-Time Coordination Protocols, one can develop a supervisory controller that guarantees the correct message-exchange between all the discrete entities that the system has. In addition to this, it should be highlighted that, in comparison with the other 3 tools/languages, MechatronicUML is the only one that provides a structured method that is always the same which basically consists of selecting both ports that will be used for the asynchronous communication and the messages that each statechart will send/receive. MechatronicUML also allows the interaction with atomic components with continuous-time behaviour (even though only through its external interface) so that the physical variables from those components can have an impact on the statecharts of the discrete components. Despite small lacks in features like the synchronization between statecharts, MechatronicUML provides with enough possibilities to develop the supervisory control structure of the controlled system.

Moving on to the validation phase, even though neither of them are performed with the same tool (and that implies problems like getting the other tools which maybe not be free to get the results), there are several ways of validation for the models created in MechatronicUML. When it comes to formal verification, a standard transformation from MechatronicUML models to the model checker UPPAAL is possible and a back translation is send in return to the developer to see if there are any failures in the system. On the other hand, two well developed transformations from MechatronicUML to Simulink&Stateflow and to Modelica are the main attraction to validate the models through simulation. The first transformation to Simulink is done directly from the MechatronicUML toolset and the second one is done with a library in Modelica which features the same functionalities than the original one. As a result, the validation phase of the systems engineering process with MechatronicUML can be well performed in different ways.

Finally, the code generation can be done directly from the MechatronicUML tool and currently only the Real-Time Java and C++ platforms are supported. It should be interesting to see the features that are included in the next versions of MechatronicUML: maybe some features for low-level modeling, more ways of validation of the models or just new platforms for the code generation. The main concern that a developer has to have with this tool is the fact that all the simulations and validation process for the low-level part of the system will have to be performed with an external tool and, if after that the validation of the overall supervisory control structure can be done with another simulation tool, maybe the modeling process should have been directly with the other tool (depending on the tool and the system of course).

8.2.3 Modelica review

The Modelica language is a modeling language with an object-oriented and equation-based approach mainly used to model complex cyber-physical systems. This is a language based on component models which are formed by several constructs that allow for the creation of physical components and the connections between them. Modelica is special because of its graphical multi-domain capabilities that allow the developer to combine physical components from several engineering disciplines and represent them with their real appearance when modeling. This differs from a tool like Simulink since the block diagram approach results in more difficulties to see the physical reality that is being modeled.

Considering all the features of the language, Modelica should be mainly used for modeling and simulating the low-level aspect of the system. It provides with a very large number of libraries that allow the developer to create very complex systems with different blocks that allow for different possibilities regarding their physical behaviour and a lot of control strategies that can be implemented either in continuous-time or discrete-time. On the other hand, when modeling the high-level of the system, Modelica has several capabilities if compared with tools like Stateflow due to simplicities in their approach towards state machines and hybrid system modeling. This is also an aspect of the language that the company is improving with any new version of the Modelica language that comes out so it will be interesting the added features that will be presented in the future. However, it should be after a few years that the Modelica language can compete with other more advanced tools when it comes to modeling the high-level supervisory control structures.

When it comes to the validation phased, the Modelica language is quite polarized in terms of the possibilities between the simulation-based visualization and the formal verification procedures. The first one is supported by using a proper simulation environment (which is also the modeling environment) like OpenModelica (which is free) or Dymola (which is not free). The capabilities between the two are quite similar. With a tool like OpenModelica, several simulation possibilities are offered so that the developer can see in a graphical way almost any aspect of the physical system or the control structure to see if the behaviour is acceptable. On the other hand, it has already been stated that because of some of the practices that are used with this language, there will be no possibilities to apply any formal methods for verification. It will be interesting to see if this changes in the future and, if so, how does the Modelica language adapt to support this formal methods. It should also be interesting to see if there will be any development of a translation procedure for Modelica methods into another tool. This lacks of formal methods for verification should also be considered in the future version of the language because if features like the state machines and the supervisory control logic development improves, there should be more possibilities to validate their behaviour and formal methods are usually the best.

Finally, the implementation phase is always related to the simulation environment. In the case of this thesis, OpenModelica provides with the possibilities for code generation for several platforms which include: C, CSharp, C++, Adevs, sfmi, XML, Java and Javascript. If a developer is thinking of using the Modelica language for MBSE, a research of the capabilities of other simulation environment like Dymola should be studied and a decision should be made in terms of it is worth to pay for that toolset to obtain better/different features than with a free tool like OpenModelica.

8.2.4 SysML review

The Systems Modeling Language is a graphical modeling language which reuses a subset of the UML 2 for defining, specifying and developing complex high-tech cyber-physical systems. The SysML modeling method/approach is quite different than the ones that the previous tools/languages use. The SysML approach is based in the 4 main aspects of the system (also referred to as pillars) which are the requirements, the behaviour, the structure and the parametrics. With this, a developer should be able to model with a high degree of detail and accuracy any cyber-physical system and its behaviour.

The modeling process is done by using a set of 9 diagrams in total which are used to design the 4 pillars of any system: the structure is defined with the block definition diagram, the internal block diagram and the package diagram, the behaviour is specified with the activity diagram, the use case diagram, the state machine diagram and the sequence diagram, the requirements are defined with the requirements diagram and the parametrics with the parametric diagram. With this modeling process, a developer can design the different aspects of the system separately and, when finished, relate them to each other to obtain the whole system with the desired functionality. This approach used by the SysML language allows to obtain models with several levels of abstraction in terms of their structure and behaviour that cannot be obtained with other languages. This

is very important when modeling huge complex high-tech systems with a very large number of components. The separation in 4 pillars also allows for independent modeling processes and therefore, alternative design can be created by changing certain aspects and addressing particular diagrams.

However, when compared to the other tools/languages that were evaluated in this thesis, the SysML also lacks in certain areas where the others do not. For example, the dynamics of the continuous/discrete time system are always static, and for static we mean that they always have the same equations just like they are defined in the constraint blocks of the parametric diagram. This reality has quite an effect when trying to build the supervisory control structure and the event-driven level of the system with state machines. The fact that it is not possible to assign variables or dynamic equations to the states of the statecharts supposes quite a drawback when trying to model systems with different operating modes where the equations/laws that define certain variables change depending on the state. This concepts lacking along the others that have been mentioned in Section 8.1 implies that SysML is (probably) not the most suitable language for supervisory controller development.

Moving on to the validation and implementation phase, since SysML is a language and not a tool, it does not provide with any sort of validation method nor code generation possibilities. However, just like with Modelica, if the engineers have the adequate platforms/tools, that is definitely not a problem since there exist transformations and modeling environment for SysML models to perform validation/verification and code generation procedures. Transformations to tools like Simulink or Modelica for simulation-based validation or the SysML based environment AVATAR which translates models to UPPAAL for formal verification are well developed and available to any engineer that uses SysML for MBSE. On the other hand, environment like the Rhapsody Designer toolset of the Enterprise Architect Systems Engineering toolset are very viable options for code generation to platforms like C, C++ or Java. Therefore, even though SysML by itself cannot provide them, with some many external possibilities available, no one that uses SysML and be scared to not be able to validate/implement the designed models.

To sum up, even though SysML might be less powerful to develop complex supervisory control structures, it is still a very powerful language for model-based systems engineering thanks to the decomposition of the systems that he does in different pillars/diagrams which allow for local design and investigating alternatives in different aspects to be integrated later as the overall system.

Chapter 9

Conclusions and Recommendations

The goal of this thesis is to evaluate and compare the capabilities in terms of model-based systems engineering with supervisory control development of modeling languages/tools Simulink&Stateflow, MechatronicUML, Modelica and SysML. To support this comparison, the Pick&Place station of the xCPS platform has been chose as a representative case study. In Section 9.1, the overall conclusions of this thesis are presented. In Section 9.2, suggestions for future work and research are made.

9.1 Conclusions

The goal of this Masters Thesis was to investigate, analyze, evaluate and compare the capabilities and possibilities of the modeling tools/languages Simulink&Stateflow, MechatronicUML, Modelica and SysML in terms their support on model-based systems engineering with supervisory control development. The xCPS platform was chosen as a case study to support the analysis/evaluation process and to try to present an example of system build with models of each tool/language.

To achieve this goal, first of all, in Chapter 2 an introduction to the topic of the model-based systems engineering process is presented to get familiar with the process and concepts that the different tools/languages will be evaluated. Next, in Chapter 3, the first case study including the whole modeling process (from specification until validation) of the xCPS platform is presented by using the CIF3 toolset. Later, in Chapters 4, 5, 6 and 7 the partial individual evaluation of each one of the languages/tools has been performed in a detailed but brief way. This evaluation is in terms of modeling, simulation, verification and implementation. In addition to this, per each language/tool a small foundation of the basic modeling concepts used by those languages/tools for modeling cyber-physical systems is provided and the case study was developed in some of the languages/tools as well. After that, in Chapter 8 the comparison between the different languages/tools has been done to see the scope of capabilities of each one and how exactly do they do against each other. In addition to this, a review section is provided to review each one of the languages/tools addressing their capabilities, main uses and possible future development. Eventually, the conclusions regarding the different modeling languages/tools are presented in this section in the following paragraphs with a summary for each modeling language/tool.

The Simulink and Stateflow toolboxes, integrated in the MATLAB environment, provides a very powerful framework for modeling and developing complex high-tech cyber-physical systems. As a result, this combination of tools is probably the one that offers more capabilities in terms of the specification phase. Just like the other languages/tools, a manual process is possible to develop the supervisory control structures that the system requires and with Simulink, a great environment for

simulation-based validation (we have to remember that formal methods for these models are still to be properly developed). Finally, code generation and implementation possibilities are usually also available (even though Simulink&Stateflow do not provide them) since the required toolboxes are usually integrated with the MATLAB toolset that each company purchases.

The MechatronicUML method was developed to mainly focus on modeling and developing the software (and communications) that are embedded in the high-tech systems that are exist nowadays. As a result, because of its nature, there exist very low possibilities when it comes to modeling the low-level time-driven side of the system since it is supposed to be developed in an external tool (like Modelica or Simulink). However, MechatronicUML provides a strong and structured method for modeling the high-level event-driven side of the system with Real-Time Statecharts for modeling the behaviour and Real-Time Coordination Protocols to develop the supervisors in a structured and compact way. When it comes to model validation, no direct possibilities are provided with MechatronicUML but, however, direct translations/transformations (well supported) exist to either validate the models through simulation or formal verification. Finally, code generation for platforms like C++ or Java are provided.

The Modelica language is an object-oriented equation-based modeling language that offers a wide variety of capabilities for modeling complex cyber-physical systems. This language supports the first two phases (specification and supervisor development) almost like the pair Simulink&Stateflow does but with less capabilities in terms of the automata formalism for the high-level event-driven of the system. It also differs from Simulink because of its graphical component-based approach used to model the hierarchical structure of systems. After modeling, to be able to validate the models and generate code to implement the controllers in a given platform, a simulation environment is required and one of the most commonly used is OpenModelica. With OpenModelica, a very powerful simulation-based environment with several analysis methods is provided to validate the models. However, it is very important to remember that with the Modelica language, because of some of its practices, formal methods for model verification do not exist. Eventually, OpenModelica offers code generation possibilities for platforms like C, C++ and Java among others.

The SysML modeling language provides a quite different approach (with respect to the other three) when it comes to modeling cyber-physical systems. Its basic approach is based in the 4 main aspects of the systems that is modeled and that are the structure, the behaviour the requirements and the parametrics. Each one of this aspects is modeled with one or more specific diagrams that allow the developer to model each aspect of the system independently allowing a lot of freedom and facilities to design and generate alternative models. However, despite this high degree of independence and detail that can be used to model a system, SysML lacks in some aspects/features (when compared to the other tools/languages) when modeling the dynamic behaviour of the system or when developing the supervisory control structures that control the the overall system. Furthermore, just like Modelica, validation and implementation possibilities are provided with external tools that support the SysML language and that provide either simulation-based or formal verification methods and further code generation and implementation to specific platforms.

9.2 Recommendations and future work

The modeling languages/tools that have been analyzed and evaluated in this thesis represent only a small portion of all the modeling tools that are available out there when trying to model high-tech cyber-physical systems. These other languages/tools, just like the ones that we studied, might have special capabilities that cannot be found in the first ones (an the other way around) so depending on the case study they might be worth being considered.

In [9], a survey of some of the languages/tools that exist for general model-based systems engineering are presented and briefly compared against each other. It is only a basic high-level comparison

but it maybe be a first step when looking for new modeling languages/tools and can give the reader quite an accurate idea of where each language is exactly at. For specific details, like the capabilities in terms of supervisory control development, further research is required. Therefore, there are a lot of tools that might be worth a try or at least some consideration when modeling cyber-physical systems. Some of these tools are:

- **CHARON** [2]: CHARON, which is an acronym for coordinated control, hierarchical design, analysis and run-time monitoring of hybrid systems, is a high-level language for modular specification of multiple interacting hybrid systems. It is based on the notions of mode and agent and it supports hierarchical modeling at the architectural and at the behavioural level. Agents model distinct components of the system whose executions are all active at the same time. Modes represent the discrete and continuous behaviors of an agent.
- **Masaccio and Giotto** [29]: Masacio is a high-level modeling language that builds complex component-based systems by using discrete/continuous atomic components and the serial and parallel composition between them. Per each component it defines an interface which contains the dependency relations which describe which continuous output signals are produced depending on which continuous input signals are used. In addition to this, the Giotto programming language provides the different models of operation and the mode switches (similar to automata models). The behaviour of a mode is described in C code and each mode is associated with a certain period which specifies how often its behaviour is executed.
- **Ptolemy II** [17]. Ptolemy is a modeling tool that, similar to CHARON, distinguishes between architectural and behavioural design when modeling. In contrast to Charon though, Ptolemy provides different semantic domains which are referred to as models of computation. These include semantics for continuous-time, discrete-time, discrete-event, finite state machines, synchronous dataflow among others. This toolset even supports the combination and integration of these models of computation.

Exploring new possibilities in terms of the modeling languages is always a viable option but it is also very important to keep an eye on the evolution of the languages that one already knows. Tools like Simulink&Stateflow and languages like Modelica are in constant development and releasing new versions of their software and therefore they might be able to perform better in certain areas where they lack at the moment. It should also be considered the modeling languages/tools that might still not be available out there but that may appear in the future and that could represent a change/step forward in what model-based systems engineering is concerned. The last thing that could be done in terms of research would be to study the integration of the languages/tools that have been studied with other modeling languages (if they support those integrations of course).

Bibliography

[1] Adyanthaya, S., Ara, H. A., Bastos, J., Behrouzian, A., Snchez, R. M., van Pinxten, J., ... & Frijns, R. (2015, October). xCPS: A tool to eXplore Cyber Physical Systems. In Proceedings of the WESE 15: Workshop on Embedded and Cyber-Physical Systems Education (p. 3). ACM. 6, 7, 8

[2] Alur, R., Grosu, R., Hur, Y., Kumar, V., & Lee, I. (2000, March). Modular specification of hybrid systems in CHARON. In International Workshop on Hybrid Systems: Computation and Control (pp. 6-19). Springer Berlin Heidelberg. ISO 690 20, 91

[3] Baeten, J. C. M., van de Mortel-Fronczak, J. M., & Rooda, J. E. (2011). Integration of supervisory control synthesis in model-based systems engineering. 6, 13, 14

[4] Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Thiele, S., Schfer, W., ... & Tichy, M. (2014). The MechatronicUML design methodprocess and language for platform-independent modeling. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Tech. Rep. tr-ri-14-337. 9, 43, 44, 45, 46, 47, 48, 49

[5] Behrmann, G., David, A., & Larsen, K. G. (2004). A tutorial on uppaal. In Formal methods for the design of real-time systems (pp. 200-236). Springer Berlin Heidelberg. 52

[6] Bernard, Y., Burkhart, R. M., de Koning, H. P., Friedenthal, S., Fritzson, P., Paredis, C., ... & Schamai, W. (2010). An Overview of the SysML-Modelica Transformation Specification. In Proc. of INCOSE (pp. 11-15). 78

[7] Bhave, A., Krogh, B., Garlan, D., & Schmerl, B. (2010). Multi-domain modeling of cyber-physical systems using architectural views. 2

[8] Brandin, B. A., & Wonham, W. M. (1994). Supervisory control of timed discrete-event systems. Automatic Control, IEEE Transactions on, 39(2), 329-342. 6

[9] Burmester, S., Giese, H., & Henkler, S. (2005, September). Visual model-driven development of software intensive systems: A survey of available techniques and tools. In Proc. of the Workshop on Visual Modeling for Software Intensive Systems (VMSIS) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC05), Dallas, Texas, USA (pp. 11-18). 90

[10] Burmester, S., Giese, H., & Schfer, W. (2005, November). Model-driven architecture for hard real-time systems: From platform independent models to code. In European Conference on Model Driven Architecture-Foundations and Applications (pp. 25-40). Springer Berlin Heidelberg. 52

[11] Carloni, L. P., Passerone, R., Pinto, A., & Sangiovanni-Vincentelli, A. (2006). Languages and tools for hybrid systems design. now Publishers Inc. 29

[12] Cassandras, C. G., & Lafortune, S. (2009). Introduction to discrete event systems. Springer Science & Business Media. 6

[13] Chen, C., Sun, J., Liu, Y., Dong, J. S., & Zheng, M. (2012). Formal modeling and validation of stateflow diagrams. International Journal on Software Tools for Technology Transfer, 14(6), 653-671. 39

[14] Clarke, E. M., Grumberg, O., & Peled, D. (1999). Model checking. MIT press. 52

[15] Cuijpers, P. J., & Reniers, M. A. (2008, April). Lost in translation: Hybrid-time flows vs. real-time transitions. In International Workshop on Hybrid Systems: Computation and Control (pp. 116-129). Springer Berlin Heidelberg. 21

[16] Cuijpers, P. J. L., Reniers, M. A., & Heemels, W. P. M. H. (2002). Hybrid transition systems. Technische Universiteit Eindhoven, Department of Mathematics and Computer Science. 21

[17] Davis II, J., Goel, M., Hylands, C., Kienhuis, B., Lee, E. A., Liu, J., ... & Smyth, N. (1999). Overview of the Ptolemy project (No. M99/37). ERL Technical Report UCB/ERL. 91

[18] Derler, P., Lee, E. A., & Vincentelli, A. S. (2012). Modeling cyberphysical systems. Proceedings of the IEEE, 100(1), 13-28. 1, 14

[19] Deshpande, A., Gll, A., & Varaiya, P. (1996, October). SHIFT: A formalism and a programming language for dynamic networks of hybrid automata. In International Hybrid Systems Workshop (pp. 113-133). Springer Berlin Heidelberg. 20

[20] Elmqvist, H., Gaucher, F., Matsson, S. E., & Dupont, F. (2012, November). State machines in Modelica. In Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany (No. 76, pp. 37-46). Linkping University Electronic Press. 57, 59

[21] Elmqvist, H., Mattsson, S. E., & Otter, M. (2014). Modelica extensions for multi-mode DAE systems. In Modelica'2014 Conference, Lund, Sweden, March (pp. 10-12). 59, 60

[22] Estefan, J. A. (2007). Survey of model-based systems engineering (MBSE) methodologies. Incose MBSE Focus Group, 25(8). 2, 5

[23] Fransen, R. Modeling a flow shop workstation using CIF. Bachelor thesis, 2015. CST 2015.077. 7, 8, 19, 20

[24] Friedenthal, S., Moore, A., & Steiner, R. (2006, July). OMG systems modeling language (OMG SysML) tutorial. In INCOSE Intl. Symp. 9, 65, 67, 68, 69, 70, 71, 72, 73, 74, 75

[25] Fritzson, P. (2014). Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach. John Wiley & Sons. 9, 55, 56

[26] Fritzson, P., Aronsson, P., Lundvall, H., Nystrm, K., Pop, A., Saldamli, L., & Broman, D. (2005). The OpenModelica modeling, simulation, and software development environment. Simulation News Europe, 44, 8-16. 55, 56

[27] Gery, E., Harel, D., & Palachi, E. (2002, May). Rhapsody: A complete life-cycle model-based development system. In International Conference on Integrated Formal Methods (pp. 1-10). Springer Berlin Heidelberg. 78

[28] Heinzemann, C., Rieke, J., Brggelwirth, J., Pines, A., & Volk, A. (2013). Translating mechatronic uml models to matlab/simulink and stateflow. Software Engineering Group, University of Paderborn, Tech. Rep. tr-ri-13-330. 51

[29] Henzinger, T. A., Horowitz, B., & Kirsch, C. M. (2001, October). Giotto: A time-triggered language for embedded programming. In International Workshop on Embedded Software (pp. 166-184). Springer Berlin Heidelberg. 91

[30] Kawahara, R., Nakamura, H., Dotan, D., Kirshin, A., Sakairi, T., Hirose, S., ... & Ishikawa, H. (2009). Verification of embedded systems specification using collaborative simulation of SysML and simulink models. Model-Based Systems Engineering, 21-28. 78

[31] Klenk, M., Bobrow, D. G., De Kleer, J., & Janssen, B. (2014, March). Making modelica applicable for formal methods. In Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden (No. 096, pp. 205-211). Linkping University Electronic Press. 62

[32] Kornai, A. (1996). Extended finite state models of language. Natural Language Engineering, 2(04), 287-290. 6

[33] Lee, E. A., & Zheng, H. (2006). HyVisual: A hybrid system modeling framework based on Ptolemy II. IFAC Proceedings Volumes, 39(5), 270-271. 20

[34] Linhares, M. V., de Oliveira, R. S., Farines, J. M., & Vernadat, F. (2007, September). Introducing the modeling and verification process in SysML. In 2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007) (pp. 344-351). IEEE. 78

[35] J.M. van de Mortel-Fronczak & M.A. Reniers. Model-Based Systems Engineering — Lecture Notes 4TC00. 2014. 3, 13, 14, 16, 17, 18, 27

[36] Object Management Group (2006-05-24). "OMG Trademarks". Retrieved 2008-02-26 4

[37] OMG SE DSIG SysML-Modelica Working Group. (2009). SysML-Modelica transformation specification. 78

[38] Osborne, L., Brummond, J., Hart, R. D., Zarean, M., & Conger, S. M. (2005). Clarus: Concept of operations (No. FHWA-JPO-05-072). 4

[39] Otter, M., rzn, K. E., & Dressler, I. (2005). StateGrapha Modelica library for hierarchical state machines. In Proceedings of the 4th international Modelica conference (pp. 569-578). 57

[40] Ouedraogo, L., Kumar, R., Malik, R., & Akesson, K. (2011). Nonblocking and safe control of discrete-event systems modeled as extended finite automata. IEEE Transactions on Automation Science and Engineering, 8(3), 560-569. 14

[41] Pedroza, G., Apvrille, L., & Knorreck, D. (2011, May). Avatar: A sysml environment for the formal verification of safety and security properties. In New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on (pp. 1-10). IEEE. 78

[42] MLA Radack, Shirley. "The System Development Life Cycle (SDLC)." Communications (2002). 5

[43] Rajkumar, R. R., Lee, I., Sha, L., & Stankovic, J. (2010, June). Cyber-physical systems: the next computing revolution. In Proceedings of the 47th Design Automation Conference (pp. 731-736). 1

[44] Rayshouny, J., Chandler, J., Sarabia, E., Sysavath, H., Ortiz, L., Wheeler, P., ... & Isaian, J. (2009). Application of model based systems engineering methods to development of combat system architectures (Doctoral dissertation, Monterrey, California. Naval Postgraduate School). 5

[45] Rosenberg, D., & Mancerella, S. (2010). Embedded systems development using SysML: an illustrated example using enterprise architect. Sparx Systems Pty Ltd and ICONIX, 4-14. 78

[46] Rumbaugh, J., Jacobson, I., & Booch, G. (2004). Unified Modeling Language Reference Manual, The. Pearson Higher Education. 2

[47] Schfer, W., & Wehrheim, H. (2010). Model-driven development with mechatronic uml. In Graph transformations and model-driven engineering (pp. 533-554). Springer Berlin Heidelberg. 86

[48] Snyder, R., Bocktaels, D., & Feigenbaum, X. (2010). Validation fonctionnelle l'aide d'une transformation SysML/Simulink. Gnie logiciel, (93), 49-53. 78

[49] Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). EMF: eclipse modeling framework. Pearson Education. 4

[50] Systems Engineering group of the Mechanical Engineering department at the Eindhoven University of Technology. Cif3. http://cif.se.wtb.tue.nl/index.html, 2015. 8, 20

[51] Szyperski, C., Bosch, J., & Weck, W. (1999, June). Component-oriented programming. In European Conference on Object-Oriented Programming (pp. 184-192). Springer Berlin Heidelberg. 43

[52] The MathWorks. Simulink® Design Verifier™ 8.7 - Users Guide, March 2016. 9, 29

[53] The MathWorks. Stateflow® and Stateflow® coder™ 8.7 - Users Guide, March 2016. 9, 29, 32, 33, 34, 35

[54] Zou, L., Zhan, N., Wang, S., & Frnzle, M. (2015, October). Formal verification of simulink/stateflow diagrams. In International Symposium on Automated Technology for Verification and Analysis (pp. 464-481). Springer International Publishing. 39

[55] https://en.wikipedia.org/wiki/Systems_Modeling_Language 65

[56] https://github.com/modelica-3rdparty/RealTimeCoordinationLibrary 52

[57] http://www.sysml.org/ 5

Appendix A

CIF3 Models

This appendix contains all the CIF3 models that were developed to build the case study of this thesis.

A.1 Plant Models of the System

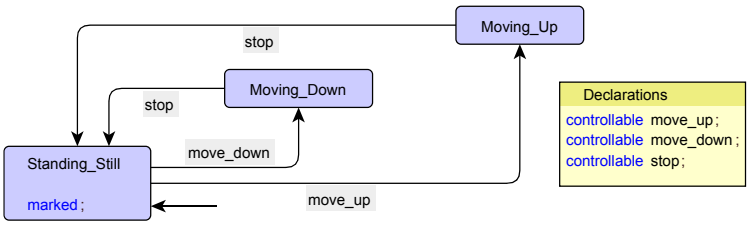

Figure A.1: Plant model of the vertical arm manipulator.

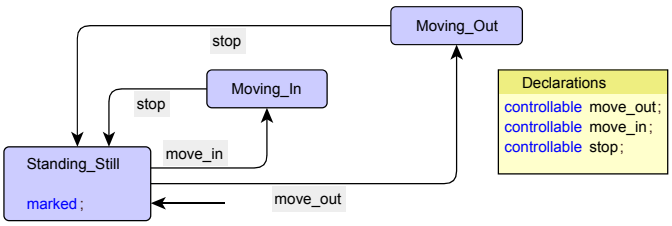

Figure A.2: Plant model of the horizontal arm manipulator.

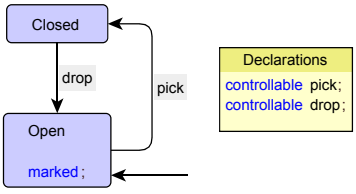

Figure A.3: Plant model of the picker.

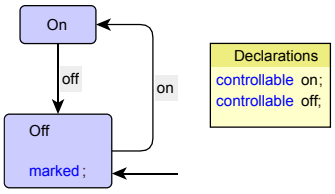

Figure A.4: Plant model of the sensors that are initially off.

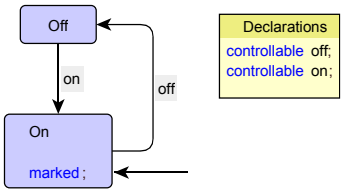

Figure A.5: Plant model of the sensors that are initially on.

A.2 Models of the requirements

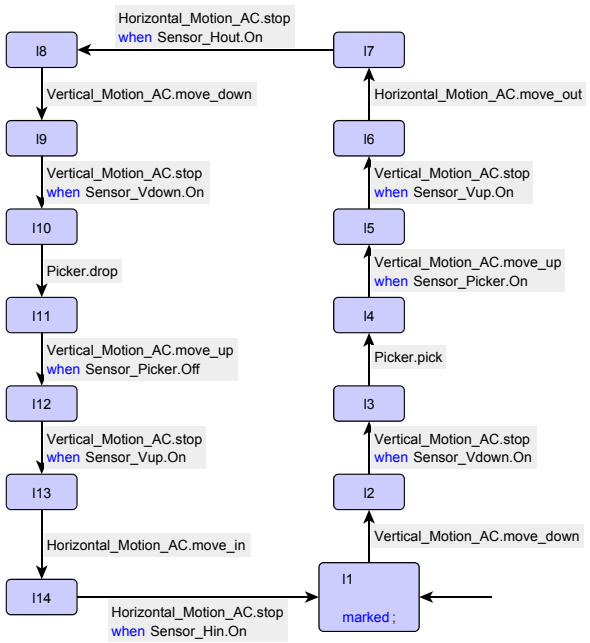

Figure A.6: Definition model of the requirements for the Pick&Place.

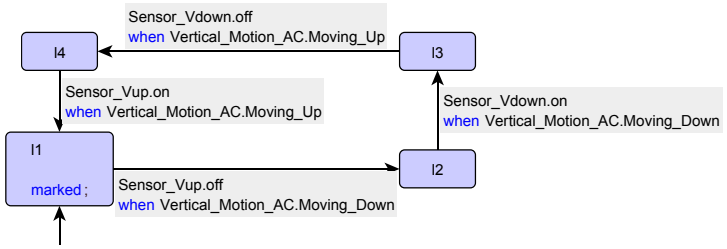

Figure A.7: Definition model of the requirements for the internal sensors of the vertical arm.

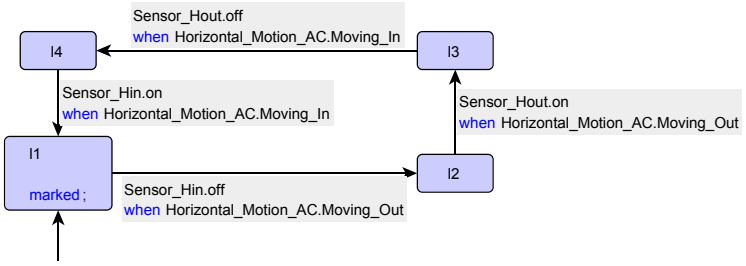

Figure A.8: Definition model of the requirements for the internal sensors of the horizontal arm.

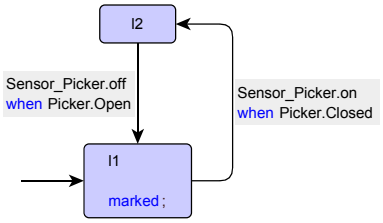

Figure A.9: Definition model of the requirements for the internal sensors of the picker.

A.3 Hybrid System Models

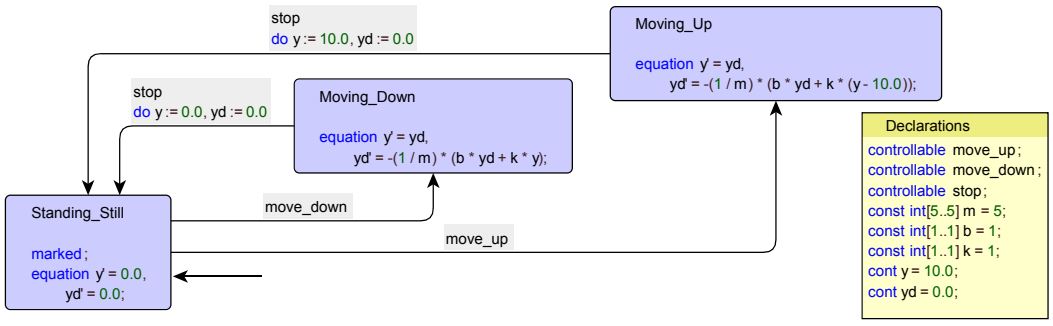

Figure A.10: Hybrid model of the vertical arm manipulator.

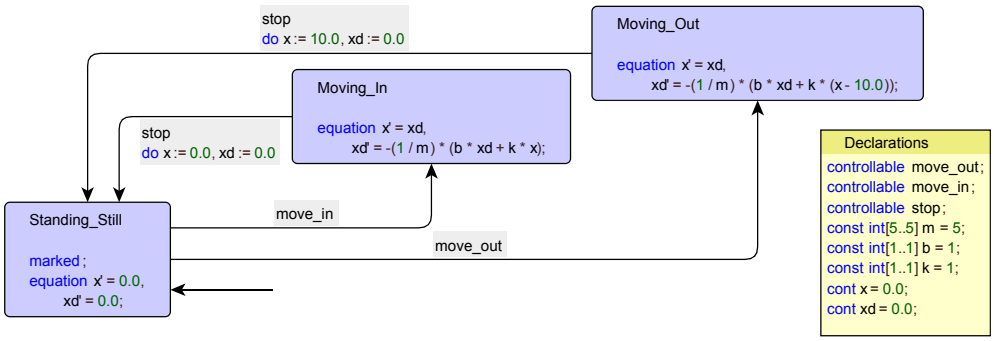

Figure A.11: Hybrid model of the horizontal arm manipulator.

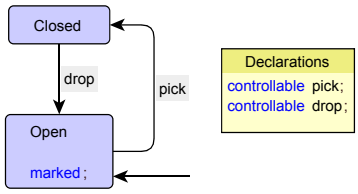

Figure A.12: Hybrid model of the picker.

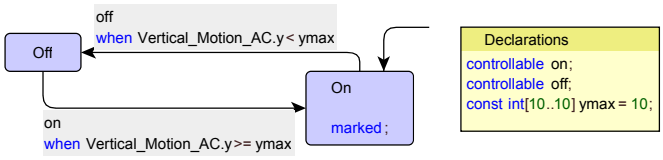

Figure A.13: Hybrid model of the internal sensor in the upper position.

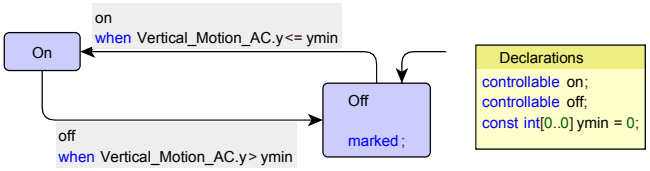

Figure A.14: Hybrid model of the internal sensor in the lower position.

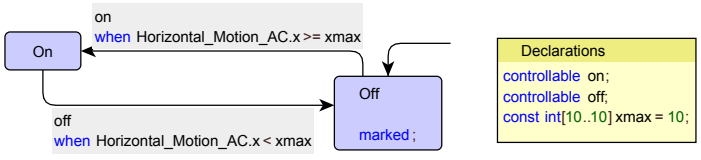

Figure A.15: Hybrid model of the internal sensor in the outter position.

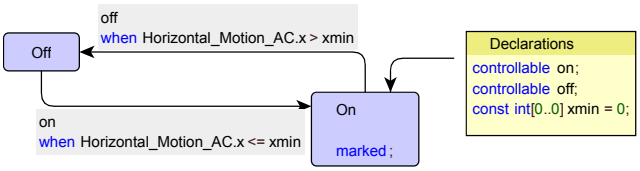

Figure A.16: Hybrid model of the internal sensor in the inner position.

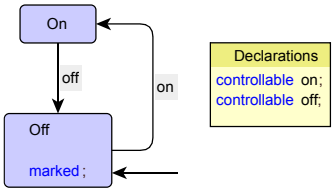

Figure A.17: Hybrid model of the internal sensor of the picker.

Appendix B

Stateflow Models

This appendix contains all the Stateflow models that were needed to build the case study of this thesis.

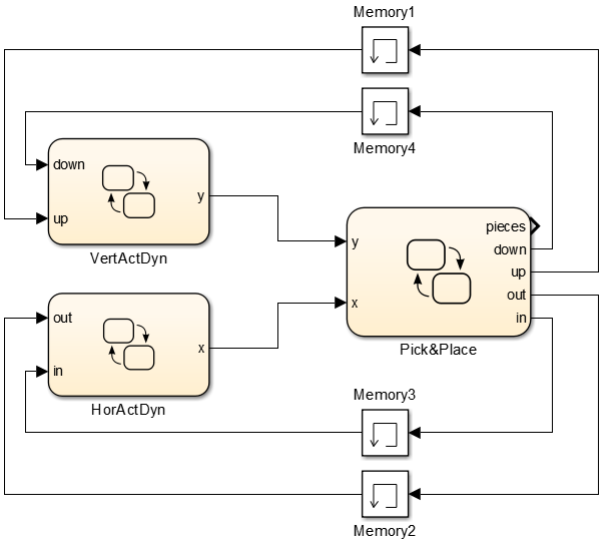

Figure B.1: Stateflow model of the Pick&Place station.

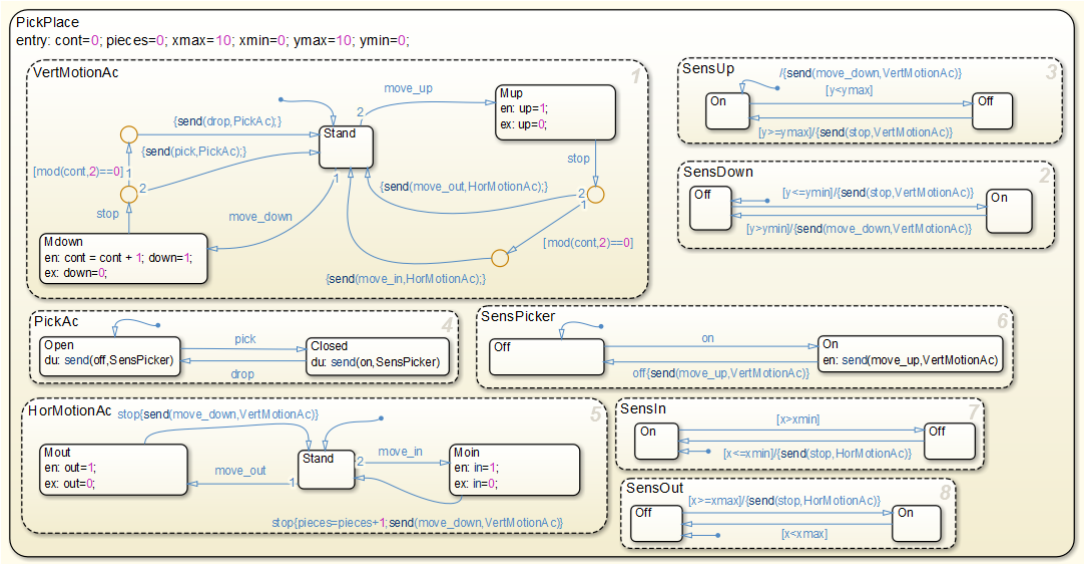

Figure B.2: Stateflow model of the Pick&Place unit without the manipulators dynamics.

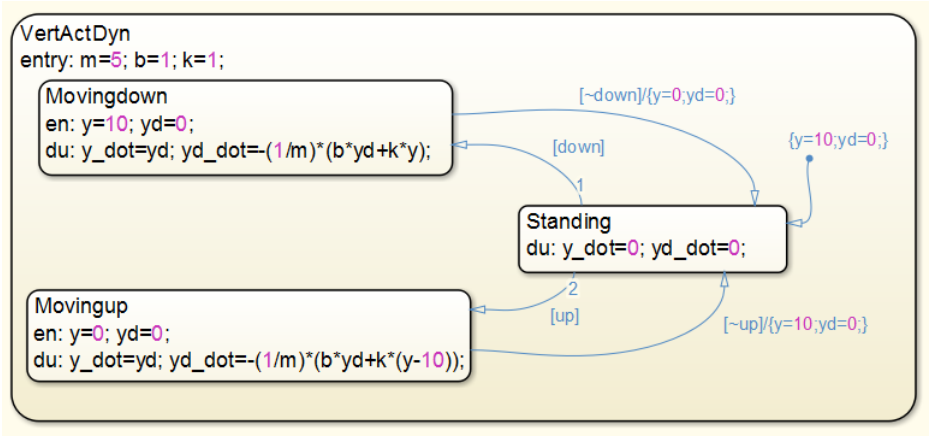

Figure B.3: Stateflow model of the dynamics of the vertical manipulator.

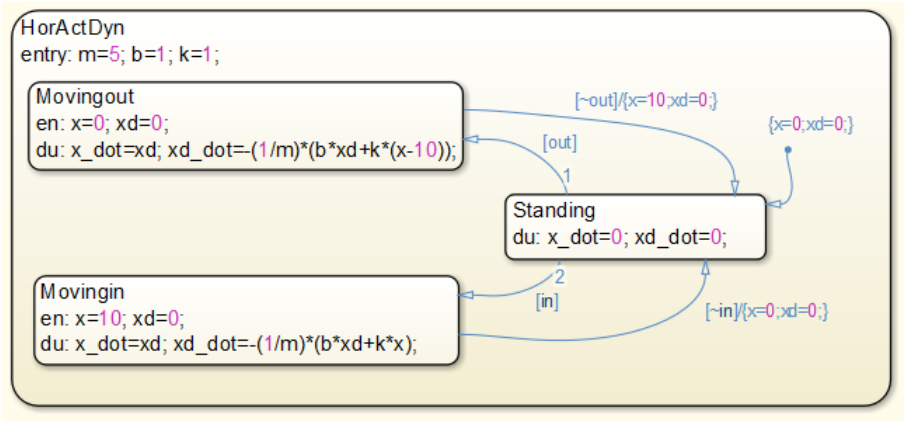

Figure B.4: Stateflow model of the dynamics of the horizontal manipulator.

Appendix C

Case study: Modelica Code

The following code has been used in this thesis to model the Pick&Place station with the model language Modelica.

```
model PickPlace

    inner Real y(start=10.0);
    inner Real yd(start=0.0);
    inner Real x(start=10.0);
    inner Real xd(start=0.0);
    inner Boolean inn(start=false);
    inner Boolean down(start=true);
    inner Boolean up(start=false);
    inner Boolean out(start=false);
    inner Boolean pick(start=false);
    inner Boolean drop(start=false);
    inner Boolean ON(start=false);
    inner Boolean OFF(start=false);
    inner Integer count(start=1);
    inner parameter Real ymax(start=10.0);
    inner parameter Real ymin(start=0.0);
    inner parameter Real xmax(start=10.0);
    inner parameter Real xmin(start=0.0);
    inner parameter Real b(start=1.0);
    inner parameter Real k(start=1.0);
    inner parameter Real m(start=5.0);
    inner parameter Boolean act(start=true);
    inner Integer res(start=0);

    block StandingStill
    end StandingStill;
    StandingStill standingstill;
    block MovingDown
      outer input Real b;
      outer input Real k;
      outer input Real m;
      outer output Real y;
      outer output Real yd;
    equation
      der(yd)=-1/m*(b*yd+k*y);
      der(y)=yd;
    end MovingDown;
    MovingDown movingdown;
    block MovingUp
      outer input Real b;
      outer input Real k;
      outer input Real m;
      outer output Real y;
      outer output Real yd;
    equation
```

```
      der(yd)=-1/m*(b*yd+k*(y-10));
      der(y)=yd;
    end MovingUp;
    MovingUp movingup;
    block Intermediate
      outer input Real b;
      outer input Real k;
      outer input Real m;
      outer output Real y;
      outer output Real yd;
    equation
      der(yd)=-1/m*(b*yd+k*y);
      der(y)=yd;
    end Intermediate;
    Intermediate intermediate;
equation
    count=if activeState(intermediate) then previous(count)+1 else previous(count);
    initialState(standingstill);
    transition(standingstill,intermediate,down,immediate=true,priority=1);
    transition(intermediate,movingdown,act,immediate=true);
    transition(standingstill,movingup,up,immediate=true,priority=2);
    transition(movingdown,standingstill,not down,immediate=true);
    transition(movingup,standingstill,not up,immediate=true);



public
    block StandingStilll
    end StandingStilll;
    StandingStilll standingstilll;
    block MovingOut
      outer input Real b;
      outer input Real k;
      outer input Real m;
      outer output Real x;
      outer output Real xd;
    equation
      der(xd)=-1/m*(b*xd+k*(x-10));
      der(x)=xd;
    end MovingOut;
    MovingOut movingout;
    block MovingIn
      outer input Real b;
      outer input Real k;
      outer input Real m;
      outer output Real x;
      outer output Real xd;
    equation
      der(xd)=-1/m*(b*xd+k*x);
      der(x)=xd;
    end MovingIn;
    MovingIn movingin;
equation
    initialState(standingstilll);
    transition(standingstilll,movingout,out,immediate=true,priority=1);
    transition(standingstilll,movingin,inn,immediate=true,priority=2);
    transition(movingout,standingstilll,not out,immediate=true);
    transition(movingin,standingstilll,not inn,immediate=true);



public
    block On1
    end On1;
    On1 on1;
    block Off1
    end Off1;
```

```
    Off1 off1;
equation
    res=rem(count,2);
    out=if activeState(on1) then res>0 else previous(out);
    inn=if activeState(on1) then res<1 elseif activeState(on4) then false else
        previous(inn);
    up=if activeState(on1) then false elseif activeState(off5) then true elseif
        activeState(on5) then true else previous(up);
    initialState(on1);
    transition(on1,off1,y<ymax,immediate=true);
    transition(off1,on1,y>=ymax,immediate=true);


public
    block On2
    end On2;
    On2 on2;
    block Off2
    end Off2;
    Off2 off2;
equation
    pick=if activeState(on2) then res>0 else false;
    drop=if activeState( on2) then res<1 else false;
    down=if activeState(on2) then false elseif activeState(on3) then true elseif
        activeState(on4) then true else previous(down);
    initialState(off2);
    transition(off2,on2,y<=ymin,immediate=true);
    transition(on2,off2,y>ymin,immediate=true);


public
    block On3
    end On3;
    On3 on3;
    block Off3
    end Off3;
    Off3 off3;
equation
    initialState(off3);
    transition(off3,on3,x>=xmax,immediate=true);
    transition(on3,off3,x<xmax,immediate=true);


public
    block On4
    end On4;
    On4 on4;
    block Off4
    end Off4;
    Off4 off4;
equation
    initialState(on4);
    transition(on4,off4,x>xmin,immediate=true);
    transition(off4,on4,x<=xmin,immediate=true);


public
    block Open
    end Open;
    Open open;
    block Closed
    end Closed;
    Closed closed;
equation
    OFF=if activeState(open) then true else false;
    ON=if activeState(closed) then true else false;
    initialState(open);
```

```
    transition(open,closed,pick,immediate=true);
    transition(closed,open,drop,immediate=true);


public
    block Off5
    //      outer output Boolean up;
    //    equation
    //      up=true;
    end Off5;
    Off5 off5;
    block On5
    end On5;
    On5 on5;
equation
    initialState(off5);
    transition(off5,on5,OFF,immediate=true);
    transition(on5,off5,ON,immediate=true);


end PickPlace;
```